REPORT DOCUMENTATION PAGE

AFRL-SR-AR-TR-04-

0527

1. REPORT DATE (DD-MM-YYYY) 08-31-2004	**2. REPORT TYPE** Final	**3. DATES COVERED** (From - To) 06-01-2001 to 05-31-2004

4. TITLE AND SUBTITLE **REAL-TIME MARBLES: A Scheme for Adaptive Distributed Resource Allocation**	**5a. CONTRACT NUMBERS**
	5b. GRANT NUMBER F49620-01-1-0341
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Dr. Martin Frank Dr. Pedro Szekely	**5d. PROJECT NUMBER**
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) USC INFORMATION SCIENCES INSTITUTE 4676 ADMIRALTY WAY MARINA DEL REY, CA 90292-6695	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) USAF, Air Force Research Laboratory Air Force Office of Scientific Research 801 N. Randolph St. Arlington, VA 22203-1977 NM	**10. SPONSORING/MONITOR'S ACRONYM(S)** AFOSR
	11. SPONSORING/MONITORING AGENCY REPORT NUMBER

12. DISTRIBUTION AVAILABILITY STATEMENT
Distribution Statement A. Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES
None

14. ABSTRACT

Purpose. The MARBLES project was established to study algorithms for coordinating swarms of Unmanned Combat Aerial Vehicles (UCAVs). Critical problems included (a) the scalability of the algorithms to very large swarm sizes, and (b) the robustness of the algorithms against the sudden total failure of participating UCAVs.
Scope. The effort focused on evaluating the algorithms in a simulated environment for the UCAVs; some of the scheduling technology was also applied to flight scheduling for AV8-B Harrier aircraft of Marine Air Group 13 in Yuma, AZ.
Methods. MARBLES used a two-pronged approach to address the critical problems above – (1) a novel peer-to-peer data storage approach among the UCAVs that scales to large swarm sizes and that is guaranteed not to lose data when any single UCAV dies, and (2) market-inspired negotiation techniques that do not exhibit a single point of failure.
Major Findings, Including Results, Conclusions, and Recommendations. MARBLES resulted in the development of a novel peer-to-peer data repository that scales logarithmically in the number of participants (UCAVs), tolerates the failure of any single UCAV without data loss, and efficiently resolves multi-attribute range queries ("locate UCAVs within this latitude and longitude with this type of munition") – and thus provided the backbone infrastructure for truly large swarms.

15. SUBJECT TERMS
scalable peer-to-peer data repositories, market-inspired negotiation, multi-agent systems, UCAV swarms, flight scheduling

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			Martin Frank
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	60	**19b. TELEPHONE NUMBER (Include area code)** 310-448-9182

20041028 082

Final Technical Report:

REAL-TIME MARBLES: A Scheme for Adaptive Distributed Resource Allocation

Grant Number: F49620-01-1-0341

Period of Performance: 06-01-2001 to 05-31-2004

Martin Frank and Pedro Szekely, Principal Investigators

Information Sciences Institute, University of Southern California

4676 Admiralty Way

Marina del Rey, CA 90292

E-mail: frank@isi.edu

WWW Homepage: http://www.isi.edu/marbles

Voice: 310-448-9182
FAX: 310-822-6592

Abstract

Purpose.
The MARBLES project was established to study algorithms for coordinating swarms of Unmanned Combat Aerial Vehicles (UCAVs). Critical problems included (a) the scalability of the algorithms to very large swarm sizes, and (b) the robustness of the algorithms against the sudden total failure of participating UCAVs.

Scope.
The effort focused on evaluating the algorithms in a simulated environment for the UCAVs; some of the scheduling technology was also applied to flight scheduling for AV8-B Harrier aircraft of Marine Air Group 13 in Yuma, AZ.

Methods.
MARBLES used a two-pronged approach to address the critical problems above – (1) a novel peer-to-peer data storage approach among the UCAVs that scales to large swarm sizes and that is guaranteed not to lose data when any single UCAV dies, and (2) market-inspired negotiation techniques that do not exhibit a single point of failure.

Major Findings, Including Results, Conclusions, and Recommendations.
MARBLES resulted in the development of a novel peer-to-peer data repository that scales logarithmically in the number of participants (UCAVs), tolerates the failure of any single UCAV without data loss, and efficiently resolves multi-attribute range queries ("locate UCAVs within this latitude and longitude with this type of munition") – and thus provided a candidate backbone infrastructure for truly large swarms.

Table of Contents

List of Figures

1 Executive Summary

The MARBLES project investigated scalable Unmanned Combat Air Vehicle (UCAV) coordination algorithms (the name of the project is not an acronym; a project member likening our particular market-inspired approach to "kids trading marbles"). These coordination algorithms address both the assignment of tasks to UCAVs without a central "leader" UCAV (Section 2) and the robust shared storage of the common knowledge (Section 3). The MARBLES project also contributed to an operational flight scheduling product for Marine Corps Harrier squadrons (Section 4). The project also resulted in two journal publications [Cai et al. 2004c, Cai and Frank 2004d], three conference publications [Cai et al. 2002, Cai et al. 2004a, Cai and Frank 2004b], five workshop publications [Frank et al. 2001, Cai et al. 2003a, Cai et al. 2003b, Decker and Frank 2004, Frank 2004], and one technical report [Cai and Frank 2003c].

2 MARBLES: Market-Inspired Negotiation for UCAV Coordination

We investigated market-inspired negotiation algorithms that are distributed in nature and that can be applied to Unmanned Combat Air Vehicles (UCAVs) as well as to other domains where decentralized decision makers must act jointly. These decision makers could act individually but are better off coordinating with their peers. A subset of this problem is distributed real-time resource allocation – deciding under time pressure which UCAVs will neutralize a newly discovered enemy threat, for example.

There is a spectrum of approaches for distributed resource allocation, ranging from no communication at all (physics-based approach: agents observe each others' behavior but do not explicitly communicate, much like a wolf pack closing in on prey) to communication of the full rationale of behavior (argumentation approach: agents back up requests to others by an argument of why they should grant it).

In this continuum, our market-inspired Marbles schemes fall in-between. Compared to a purely physics-based approach, they obviously use more messages yet also explore a more complex set of alternatives. Compared to the argumentation approach, they exchange messages of smaller complexity, yet the prices set by supply and demand can possibly communicate rationale in an alternative, more compact fashion, and potentially steer the group of agents to sensible behavior via "the invisible hand of the market".

2.1 External Marbles Scheme Properties

"Marbles" schemes are a family of resource allocation algorithms that are characterized by the following properties:

Distributed. Each task only knows about its local requirements, and communicates with potential resources for those requirements exclusively through messages. Hence, each task and each resource can – but does not have to – be located on a different machine.

Cooperative. Marbles schemes are not designed to tolerate malicious participants, which distinguishes our research from work on e.g. electronic commerce and automated auctions; we believe that security against external attack of cooperative negotiation schemes is best located at a lower level (such as the message transport and encryption level). The cooperative nature of the negotiation also means that tasks participating in resource auctions can altruistically commit suicide by permanently withdrawing, and therefore lowering resource prices that possibly help others succeed. The distributed algorithms for concluding that tasks are unlikely to succeed further distinguishes our work from work on competitive auctions.

Adaptive. A Marbles scheme can adapt a current partial solution to a new situation rather than having to re-compute the new solution from scratch. This makes them applicable in cases where "the world can't stop while a solver computes a solution for everyone", that is, in cases where the time interval between situation changes is smaller than the total running time of a non-adaptive centralized solver.

Real-Time. The individual negotiation participants should be explicitly aware of time and adapt their behavior based on how much time is left.

Fault-Tolerant. A Marbles scheme should be robust against a set level of message loss, in the sense of being able to make statements like "given an average message delay of 2 seconds and a message loss rate of 5%, this negotiation has a 99% likelihood of concluding in less than 3 minutes". Obviously, no message-based scheme can ever be robust in the sense of making a 100% real-time response guarantee if there is a non-zero chance of a message getting lost.

2.2 Internal Marbles Scheme Properties

We further characterize Marbles scheme by their "internal" properties; that term is accurate in the sense of being more linked to our approach than the above "external" ones.

Domain-based task valuation. Marbles schemes put a value on the execution of tasks that is quantitative and that has meaning to domain practitioners. The value of resources is exclusively derived from the value of the tasks they enable; they have no intrinsic domain value of their own.

Lack of inflation. We do not allow inflation (the artificial introduction of currency not backed up by domain value during negotiation) because the overall solution can otherwise not be verified in domain terms. For example, imagine that a negotiation scheme introduces inflation by increasing the value of tasks the longer they go unfilled during negotiation: consequently, the basis of the proposed overall solution cannot be analyzed by a domain expert without understanding the negotiation algorithm. Thus, the problem with mixing intrinsic task value and negotiation-scheme-dependent artificial "value" is that it would make the term "domain" currency meaningless.

Ever-fluctuating prices. In the prototypical open-outcry auction, participants bid until no one wants to bid higher, and the highest bidder then owns the resource from that point in time on. In contrast, Marbles schemes resources continually auction themselves -- the auctions never "close". That is, you can only be the current, not final, winner, of a resource -- if the situation changes because, e.g., a new high-valued task appears you will lose it.

2.3 Formal Problem Statement

Below we introduce the minimalist problem statement that our existing Marbles schemes operate on.

Problem

There is a collection of available resources that are characterized by a unique name (and nothing else). There is a collection of possible tasks that are worth a fixed domain value if they are executed. They need to acquire one resource for each of their requirements to be executed. Each resource can only be used for at most one task. Each task knows in advance which resources are suitable for its requirements. (Thus, a prior "resource discovery" phase is outside the scope of this project.)

This problem is very complex if tasks have multiple requirements ("complementaries" exist, in economic jargon) - it would be trivial if each task had just a single requirement.

Solution

A solution consists of an assignment of resources to requirements such that every task has either none or all of its requirements filled. The quality of a solution is measured by the sum of the domain values of its satisfied tasks; a higher sum indicates a better solution.

2.4 Running Example

We will use the following example depicted in Figure 1 to explain how the various Marble scheme variants operate. There are four resources called A, B, C, and D. There are two tasks called Q and R of domain value 300 and 100, respectively. Each of the two tasks has two requirements that can be filled by the resources indicated with a triangle. This particular example was chosen because it is small yet leads to backtracking behavior if schemes assign resources to requirements from left to right (as they usually do). The optimal solution of domain value 400 is obvious (Q gets A and D, R gets B and C).

		A	B	C	D
Q	1	▲	▲		▲
(300)	2	▲			
R	1	▲		▲	
(100)	2		▲		

Figure 1: The running example problem

2.5 A Rough Taxonomy of Solvers

We will present a number of "solvers" – any piece of code that produces a solution given a problem in the above terms. Our research interest is exclusively in fully distributed resource allocation schemes, but we have also built a number of centralized solvers for comparison purposes. In addition, some of our Marbles variants have so far only addressed part of the challenge in a distributed way.

All Marbles solvers fundamentally perform two tasks: assigning resources to the highest-bidding tasks ("allocation"), and eliminating tasks from competition ("elimination") because they drive up the prices for others without seeming to have a chance of obtaining all of their needed resources. Each of the variants below indicates if it solves each phase in a distributed or centralized fashion.

Marbles2 [*allocation*: distributed, *elimination*: centralized]

The main inspiration behind this Marbles variant is that the cost of a resource should be defined by the value that the second-highest bidder places on it (the "displacement" or "opportunity" cost of the resource). Consequently, resources cost zero if no one else wants them.

Message Protocol

Task to resource: bid(amount), withdrawal(); resource to task: loss(), win(amount that can be lower than bid), priceChange(can be up or down but recipient is still winning).

The Running Example under Marbles2

In this variant of our Marbles schemes, tasks attempt to fill each requirement one at a time, bidding all of their available value to satisfy the next unfilled requirement.

1. Q simultaneously bids 300 on A, B, and D to satisfy its first requirement. R bids 100 on A and C.

		A	B	C	D
Q	1	▲300	▲300		▲300
(300)	2	▲			
R	1	▲100		▲100	
(100)	2		▲		

Figure 2: First stage of marbles2 solution to the problem

2. Q obtains A for 100 (the cost as a displacement cost is determined by the second highest bidder). It reacts by bidding 200 for B and D (because it has internally determined that it is better off by using A for its second requirement, and has already spent 100 of its 300 value for obtaining a resource.

		A	B	C	D
Q	1	▲100!	▲200		▲200
(300)	2	▲			
R	1	▲100		▲100	
(100)	2		▲		

Figure 3: Second stage of marbles2 solution to the problem

3. R wins C for 0 (as there are no competing bidders). It reacts by bidding 100 on B to obtain its second resource, and by completely withdrawing its bid for A (it already has a resource for its first requirement for free; otherwise it would have bid on A whatever it had to pay for C minus the minimum bid increment/decrement).

4. Q gets notified that the price of its A dropped to 0 (because all competition disappeared). It thus now increases its bids for B and D to 300. Exclamation marks indicate that the task is currently winning the resource.

		A	B	C	D
Q	1	▲0!	▲300		▲300
(300)	2	▲			
R	1	▲100		▲0!	
(100)	2		▲		

Figure 4: Third stage of marbles2 solution to the problem

5. Q wins B for 100 (because that's R's bid). It is now satisfied, but bids 99 for D (a cheaper resource is always preferable) just in case.

6. Q wins D for 0 because no one else wants it. It withdraws its bid for B because nothing beats a free resource.

7. R gets notified that it is now the winner on B (also for 0). The scheme is in a terminal state unless the environment changes (new high-value tasks could steal resources, for example).

		A	B	C	D
Q	1	▲0!	▲		▲0!
(300)	2	▲			
R	1	▲		▲0!	
(100)	2		▲0!		

Figure 5: Final stage of marbles2 solution to the problem

Thus, in the end it has been determined that there is no competition for resources at all – all tasks can be satisfied with the available resources, using about 12 messages overall and about 4 message round-trips.

Experience and Limitations of Marbles2
As is evident from the curves in the Evaluation section below, this Marbles scheme (the first one written) tends to produce the lowest-quality solutions and also require largest number of messages. We believe that the latter is true because tasks bid on all qualified resources for every requirement, and in addition the scheme bids down prices one by one in epsilon increments (rather than in logarithmic sizes as some of the schemes below do). We have not had the time to investigate why the former is true.

Msmarbles [*allocation*: distributed, *elimination*: distributed]

In the Msmarbles (Multi-Sized Marbles) scheme each task has the same number of marbles. The size of each marble is the total value of the task divided by the number of marbles that the task has. Consequently, tasks with higher value have larger marbles.

Message Protocol
Tasks bid on resources by placing marbles on them. A task can bid one marble at a time, and must wait for a price-update message from the resource before placing another marble. Resources grant themselves to the task that has placed the largest value (not largest number) of marbles on them. When a task runs out of marbles, it can withdraw its marbles from a resource. When it does so, the resource returns all marbles to all tasks that have bid on it, with one exception. The resource keeps one marble from the current winner. In essence, the price for the current winner goes down to one marble.

When a task withdraws its marbles from a resource, it will not attempt to bid on that resource again unless it has available at least one more marble than it got back. We call this number of marbles the task's "block amount" on a given resource. Block amounts always go up, and eventually will reach the point where a task cannot win an allocation of resources for all its requirements because the block amounts on the required resources exceeds the total number of marbles that a task has. When this happens, the task voluntarily withdraws from competition by withdrawing all marbles from all resources. The scheme converges because tasks keep withdrawing until all remaining tasks succeed.

The intuition behind Msmarbles is that if the valuation of resources emerges incrementally, in small steps, it will be more accurate. This will enable tasks to make more informed decisions about where to place or withdraw marbles and when to give up, and thus lead to a better solution.

The timing of withdrawals is critical. It is advantageous to delay withdrawals as long as possible because by that time other tasks may have withdrawn first and hence they become subject to the eventually deadly block-amounts. In order to diminish the advantages of delays, we made each task have the same number of marbles, each task bid a single marble at a time, and each task wait for a reply before bidding the next marble. Richer tasks will have an advantage, as they should, because they can delay placing marbles. Poorer tasks may need several bids to catch up to the bid of a richer task, hence allowing the richer task to hold on to its marbles for a longer time.

One of the main qualities of Msmarbles is that multiple medium-sized tasks can together bid up the valuation of multiple resources forcing a richer task to become subject to several block amounts, and eventually forcing it to give up. This enables the Msmarbles scheme to make trade-offs between multiple medium-sided tasks and few richer tasks.

The Running Example under Msmarbles
Figure 6 shows the behavior of Msmarbles in the simple running example. In this example we gave each task 8 marbles (twice the number of resources). Task Q's marbles are worth 37.5 points, whereas Task R's marbles are worth 12.5 points. Lines labeled A, B, C and D represent the valuation of resources A, B, C and D over time. Lines Q-A, Q-B and Q-C represent the amount task Q has bid for resources A, B and C. R-A, R-C and R-B represent task R's bids. Initially, both tasks bid on A. Then they bid on the next resource they need: Q bids on B and R bids on C. When responses come back, Q learns that it is winning both resources. Task R learns that it is losing on A and winning on C. Task R must now bid for B, its only choice for requirement 2, and it keeps placing marbles on it until it outbids task Q. When Q is outbid it determines that the price increment to win D is 0+ (i.e., any amount larger than 0), and hence places a marble on D. At this point, both tasks are fulfilled and they stop bidding.

The second graph shows a more complex example where not all tasks can be fulfilled, and tasks need to withdraw bids and eventually withdraw from competition. The graph shows the evolution of the price for resource A, the amount task R bids on A (R-A), and the amounts task S bids on resources A, B and C (S-A, S-B and S-C respectively). In this example there are 4 tasks and 8 resources (not all shown in the graph), and S is the poorest task with 60 points. The graph shows how S first went on a bidding war for resource C and eventually withdrew because it needed marbles to bid on other resources. Similarly, S had losing bidding wars for resources A and B. A, B and C were the only choices that S had to fulfill one of its requirements, and after the three withdrawals, the block amounts went so high that S would have had to use all its marbles to win one of those resources, leaving no marbles to win resources for its other requirements. At that point, S gave up, enabling the other three tasks to succeed. The price for A went down sharply enabling the task that needed it to use its marbles for other resources. (The second problem comes from an example that Walsh uses to demonstrate that simple auctions cannot be used to compute optimal resource allocations when complementarities are present. For this particular example -- but by no means for all -- Msmarbles computes the optimal solution).

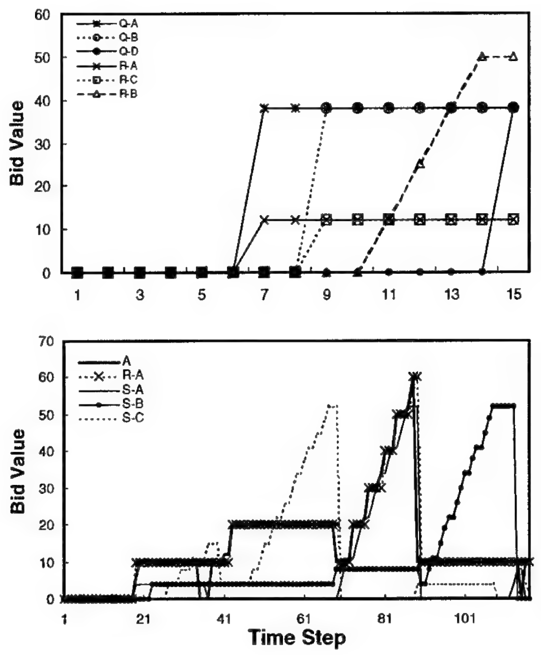


Figure 6: Bid values sequences for the Msmarbles scheme

Experience and Limitations of Msmarbles

The Msmarbles algorithm has not been as thoroughly evaluated as the others, so that implementation bugs disqualify it from the systematic comparison with the other algorithms in the Evaluation section. The solutions of the examples it does run are of high quality (defined as "close to the best solutions of other schemes"). However, the scheme is also one of the slowest, using significantly more messages than the others.

Marblesize [*allocation*: distributed, *elimination*: distributed]

The motivation of the Marblesize scheme is to allow trading off the quality of the solution against the number of messages needed through different pre-specified Marbles "sizes".

In the Marblesize scheme, no resource is free and the price for a resource is determined by the current highest bid. To acquire a resource, a task needs a certain number of marbles. Marbles have given size that can be subdivided in equal parts an arbitrary number of times. The size of the marbles represents the minimum amount a task can bid on a resource. For each task, the initial marble size is equal to the task value divided by the number of requirements in that task. At the beginning, each task selects a possible combination of resources for its requirements and bid one marble on each of them. After that the bidding mechanisms continues based on the following rules: (1) If a task has more than one possible combination of resources, it chooses the cheapest one based on the current bids on those resources and allocate all its value among them but placing at least one marble on each resource. (2) A task wins if it is winning on all of its current bids. (3) A task loses if it is losing on all of its current bids. (4) A task that is winning on some of its bids can move one marble at a time from a winning resource bid to a losing resource bid. (5) A task can cut its marble size until the marble size is less than the minimum marble size allowed. (6) A losing task tries another resource combination and repeats the process. If it cannot find a new combination of resources it commit suicide.

Message Protocol

Task to resource: bid (amount), withdrawal (); Resource to task: loss (), win ().

The Running Example under Marblesize

First round: Q: Marble size (150) R: Marble size (50).

		A	B	C	D
Q (300)	1	▲	▲150!		▲
	2	▲150!			
R (100)	1	▲50		▲	
	2		▲50		

Figure 7: First stage of marblesize solution to the problem

The two requirements of task Q are winning so no changes happen in that task. In task R both requirements are losing. Since the first bidding proposal is no good it tries a second bidding proposal [C,B] while keeping a marble size of 50.

Second round: Q: Marble size (150) R: Marble size (50).

		A	B	C	D
Q (300)	1	▲	▲150!		▲
	2	▲150!			
R (100)	1	▲		▲50!	
	2		▲50		

Figure 8: Second stage of marblesize solution to the problem

Now R is winning on C that nobody wants and tries to move its marbles from C to B.

Third round: Q: Marble size (150) R: Marble size (25).

R cuts it marble size to the minimum size of 25. Although R is still winning on C, it cannot move its marble anymore because each resource needs at least one minimum size marble. So R's second proposal is declared dead. Since it cannot try a third proposal, R is declared dead and the process terminates. (Thus, the scheme fails to find the optimal solution for this simple problem - nevertheless it is the single best scheme we have for large problems, as will become evident in the Evaluation section.)

		A	B	C	D
Q (300)	1	▲	▲150!		▲
	2	▲150!			
R (100)	1	▲		▲25!	
	2		▲75		

Figure 9: Final stage of marblesize solution to the problem

Experience and Limitations of Marblesize
The Marblesize scheme has the unique ability to trade off solution quality against speed of convergence. Figure 10 shows the impact that the minimum marbles size has on the total number of messages and the quality of the solution. As is evident, it is possible to control the minimum marble size to trade-off solution quality for computational time. In this example, an increase of less that 1% of solution quality is paid by a 10 fold increase in the number of messages.

Number of Subdivisions	Total Number of Messages	Maximum Value of Solution
4	7986	19017
3	6013	18950
2	3635	18894
1	1250	18880
0	793	18649

Figure 10: Trade-off between message traffic and solution quality in msmarbles

In terms of scalability with respect to problem size, the number of messages and solving shows a phase transition behavior where, for fixed number of resources, the number of messages increases sharply with the number of tasks until it reaches a certain value where starts decreasing again, resembling the critical behavior observed in other combinatorial problems. We believe that this is due to the fact that for large number of tasks the lack of resources leads to quick suicide of most tasks with large requirements, thus the competition quickly decreases along the process.

Grabmarbles [*allocation*: distributed, *elimination*: distributed]
Grabmarbles is a variation of the Marblesize scheme which relies on heuristic selection of resource combinations. As in Marblesize, a task bids on the cheapest set of resources that will satisfy its requirements. Unlike the Marblesize scheme, rebidding is not permitted after a losing resource bid, and bids are not based on marble sizes. Instead, a task agent submits a bid that is a heuristic evaluation of the task, based on its domain value, number of task requirements, and number of alternative resources. A task only bids for resources whose prices (the evaluations of the currently winning tasks) are less than the bidding task's own evaluation. When a task agent loses a bid, it gives up on the current resource set and tries another if possible. The heuristics used by Grabmarbles were originally applied to Marbles2, and improvements in solution quality motivated the

application of those heuristics to Marblesize.

A heuristic task evaluation function is defined for a given task and resource. (Note that this heuristic function actually violates the "no inflation" rule for Marbles schemes, making it impossible to use the prices paid for resources as an indication for ther contribution of domain value. This has not been an issue because we have only measure pure solution quality so far.) The following example of a task evaluation function rewards tasks that have only one or two alternative resources to choose from, otherwise penalizing the task according to its number of requirements.

```
function taskeval (dval, reqs, alts)
  if alts = 1 return dval / reqs;
  else if alts = 2 return dval / (2 * reqs);
  else return dval / (4 * reqs);
```

Message Protocol
Task to resource: bid (amount), withdrawal (); Resource to task: loss (), win ().

The Running Example under Grabmarbles
First round: Q selects A and B.

		A	B	C	D
Q (300)	1	▲	▲37.5!		▲
	2	▲150!			

Figure 11: First stage of grabmarbles solution to the problem

The running example is analyzed here using the task evaluation function described above. All resources are initially free, so task agent Q selects A and B. Q's domain value of 300 and its 2 requirements yield an evaluation of 37.5 for resource A, while its evaluation with respect to A (150) reflects the fact that A is Q's only alternative resource for requirement 2.

Second round: R selects B and C.

		A	B	C	D
R (100)	1	▲		▲25!	
	2		▲50!		

Figure 12: Second stage of grabmarbles solution to the problem

The possible resource sets available to task agent R are (A,B) and (C,B). The cheaper alternative is (C,B), whose total price of 37.5 is due to Q's currently winning bid. Like task Q, R's second requirement has only one qualified alternative, so R's task evaluation with respect to resource B comes to 50. Task Q is outbid for resource B, so it withdraws its bids and tries another resource combination.

Third round: Q selects A and D.

		A	**B**	**C**	**D**
Q (300)	1	▲	▲		▲37.5!
	2	▲150!			
R (100)	1	▲		▲25!	
	2		▲50!		

Figure 13: Final stage of grabmarbles solution to the problem

Task agent Q finally selects price-free resources A and D. Both tasks are now satisified, reaching the optimal solution domain value of 400, with 15 messages passed.

Experience and Limitations of Grabmarbles

The Grabmarbles scheme produces solutions that are comparable to those of Marblesize, with a relatively small number of messages. The use of heuristics in evaluating each task's "deservedness" with respect to different resources has a globally beneficial effect on resource allocation. In the Marblesize scheme, the relative merit of competing tasks is resolved through the process of rebidding and transferral of funds between resources. In Grabmarbles, the selection of resources through heuristics tends to direct the task agents toward resources they can realistically attain, while avoiding resources that are critical to other tasks. The focus on globally beneficial resource selection helps to eliminate the need for rebidding.

The choice of task evaluation formula used in Grabmarbles has not yet been automated. The quality of solutions is greatly affected by how well suited the evaluation formula is for a particular problem set. The results shown in the curves in the Evaluation section were obtained using the following evaluation function.

```
function taskeval (dval, reqs, alts)
  return dval / reqs – 2 * alts;
```

This evaluation formula fails to yield the optimal solution domain value for the running example problem. The previous formula emphasizes the lack of resources available to a task, while the above formula only uses this as a tie-breaker. A hybrid evaluation formula, combining features of the two shown, has produced good solutions to all of these problem sets. But there remains a need for the automatic selection of an approriate formula for a given problem, based on the distribution of task requirements per task, and alternative resources per requirement.

Brute-Force [*allocation*: centralized, *elimination*: centralized]

We have built a trivial centralized brute-force solver that enumerates all possible solutions and then picks the best one. It is impractical for more than about 15 tasks and 30 resources but serves its purpose in producing small-size challenge problems for the Marbles schemes for which the optimal solution is known.

Random [*allocation*: centralized, *elimination*: centralized]

Similarly, we have built a solver which synthesizes a random solution, keeps it if it beats the previous one, and keeps doing this until it exceeds a given time limit. We have used it to establish lower bounds on the solution quality for large-size problems.

Simulated Annealing [*allocation*: centralized, *elimination*: centralized]

We have implemented a Simulated Annealing (SA) solver [Kirkpatrick 1983] to further compare the results of the different Marble solvers against well-known central schemes. The SA algorithm seeks to escape local maximum by accepting downhill moves with a probabilistic model based on statistical mechanics. In our implementation of SA we start by randomly assigning resources to tasks until all resources are allocated. Then, for a number of maxFlips times, we perturb or flip the state of the system to a neighboring state by randomly picking a task, a requirement from that task and a new resource for that requirement from its list of eligible resources. We evaluate δ, the change in the total value, and always accept the move if $\delta \geq 0$. If $\delta < 0$, we accept the move with probability $\exp(\delta/T)$, where T is the temperature parameter. We repeat this procedure for different values of T, starting with a high value of T and decreasing it following a geometric scheduling such that Ti+1= 0.5*Ti.

Experience and Limitations of the SA implementation

In terms of performance the SA solver ranks very close to but actually below the Marblesize solver. In certain problem instances SA beats Marblesize in finding a higher value in comparable execution size but on average Marblesize beats SA. SA provides the maximum number of flips (maxFlips) as its mechanism for externally controlling or trading-off quality of solution for execution time, similar to Marblesize using marble granularity for the same purpose. Even for a surprisingly low values of maxFlips, SA finds solutions within a few percent of the highest value with a significant speed up in solution time. With such a low value of maxFlips, SA is our "most efficient" solver (as measured by dividing solution quality by running time).

SAT Encoding [*allocation*: centralized, *elimination*: centralized]

We have also implemented a centralized SAT solver by encoding the resource allocation problem into Boolean satisfiability formulas in conjuctive normal form (CNF). In this approach, the allocation of resources to tasks is obtained by finding truth assignments to the resulting formulas. To use satisfiability testing for optimal allocation of resources we turn to the problem of finding valid assignments of resources for at least k (with $k \leq N$, the total number of tasks) tasks and then do a binary search to find the maximum k. This problem can then be encoded into a CNF formula of the following form:

$$f = f_k \wedge f_{cross} \wedge f_1 \wedge f_2 \wedge .. \wedge f_N$$

Where fk is responsible for switching on at least k of the variables representing the N tasks, fcross precludes resources from being assigned to more than one requirement and fi (i=1,2,…,N) selects eligible resources within each individual task.

SAT encoding of the running example

To encode the running example presented above for at least two tasks (k = 2) filled we define the following 13 boolean variables. First we introduce the tasks variables: t1 and t2, that represent each task in the formula. Then we define the resources variables A11, A12, A21, B11, B21, C21 and D11. Where Aij=TRUE indicates the assignment of resource A to task i requirement j. To select at least 2 different tasks variables we introduce four additional variables p1, p2, r1, r2 with the condition that p1 → -r1, p2 → -r2, (p1,r1) → t1 and (p2,r2) → t2. With this variables definition, the formulas introduced above take the following form:

$$f_{k=2}=(p_1 \vee p_2)\wedge(r_1 \vee r_2)\wedge(\bar{p}_1 \vee \bar{r}_1)\wedge(\bar{p}_2 \vee \bar{r}_2)\wedge(\bar{t}_1 \vee p_1 \vee r_1)\wedge$$
$$(t_1 \vee \bar{p}_1)\wedge(t_1 \vee \bar{r}_1)\wedge(\bar{t}_2 \vee p_2 \vee r_2)\wedge(t_2 \vee \bar{p}_2)\wedge(t_2 \vee \bar{r}_2)$$

$$f_{cross}=(\bar{A}_{11} \vee \bar{A}_{12})\wedge(\bar{A}_{11} \vee \bar{A}_{21})\wedge(\bar{A}_{12} \vee \bar{A}_{21})\wedge(\bar{B}_{11} \vee \bar{B}_{22})$$
$$f_1 = (\bar{t}_1 \vee A_{11} \vee B_{11} \vee D_{11}) \wedge (\bar{t}_1 \vee \bar{A}_{11} \vee \bar{B}_{11}) \wedge (\bar{t}_1 \vee \bar{A}_{11} \vee \bar{D}_{11}) \wedge$$
$$(\bar{t}_1 \vee \bar{B}_{11} \vee \bar{D}_{11}) \wedge (\bar{t}_1 \vee A_{12})$$
$$f_2 = (\bar{t}_2 \vee A_{21} \vee C_{21}) \wedge (\bar{t}_2 \vee \bar{A}_{21} \vee \bar{C}_{21}) \wedge (\bar{t}_2 \vee B_{22})$$

We solve the resulting formula f using a Java implementation [Jackson] of the WSAT [Selman 1993] solver. One can verify that f evaluates to TRUE by setting A12, B22, C21, D11, t1, t2, p1 and r2 to TRUE and all other variables to FALSE, which yields the correct solution for the running problem.

Experience and Limitations of the SAT Encoding

Our current SAT-based solver performs very well compared to Marblesize and Simulated Annealing for small and medium size problems (i.e., N ≈ 50). For larger problems (e.g., N=100) the solution time degrades about an order of magnitude compared to Marblesize and SA but it is still able to produce high value results. By controlling the number of solutions that we ask WSAT to generate, we can externally trade-off solution quality with execution time and the solver can sometimes find solutions within less than 5% of the best value found with SA but with 10 to 20 times speedup. Another advantage of this approach is that it can be used to rapidly estimate the maximum number of filled tasks without having to search for the optimal solution. In its current implementation the SAT-based solver is fully centralized but the same SAT encoding approach can be combined with Marbles or other distributed market mechanisms [Walsh 1998] to produce a distributed solver.

2.6 *Evaluation of the MARBLES Algorithms*

We evaluated the performance of the different solvers described above on synthetic problems that have the same characteristics of the problems stated above but with arbitrary number of resources and tasks. The problems were generated by randomly assigning to each task a certain number of requirements and a task value. The set of possible resources for each task was also randomly selected from the original resource pool. These random values were independently selected from three different Gaussian distributions. Thus, the dominant parameters in describing a given problem are: a) number of tasks, b) number of resources, c) r, average number of requirements per tasks, d) v, average task value and e) p, the average number of possible resources per requirement.

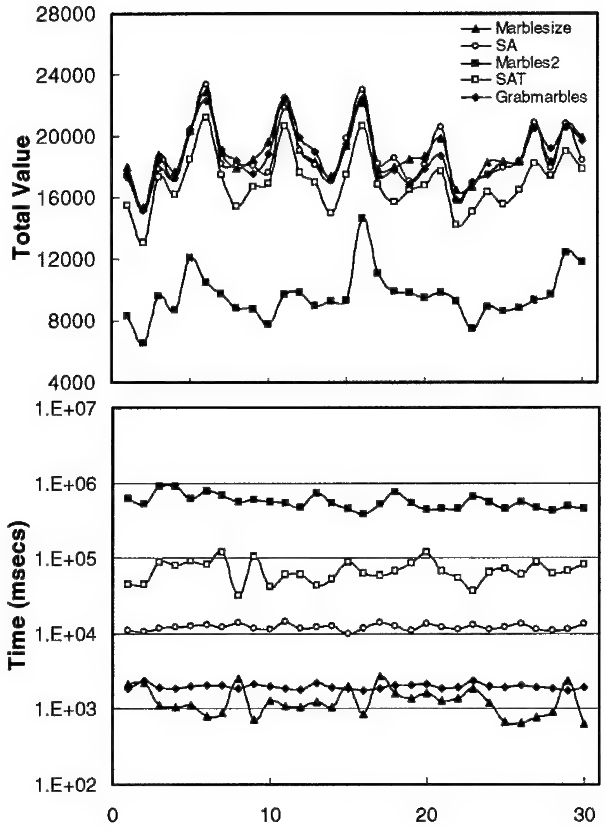


Figure 14: Quantitative Comparison of MARBLES Algorithms

In Figure 14, we compare the performance of our solvers for 30 different problems with 100 resources and 100 tasks. (The problems were generated with r =4, v = 300 and p = 10.) The parameters we use to evaluate performance are the total value (i.e., the sum of the task values for all filled tasks) of a solution and the (execution) time it took the solver to find that solution. In Figure 1a and b, we compare the results for total value and time, respectively. We see that Marblesize, Grabmarbles and Simulated Annealing can find comparable results of the total value but with Marblesize being 3 to 4 times faster than Grabmarbles and about an order of magnitude faster than Simulated Annealing. The results obtained with SAT and Marbles2 are of lesser quality in terms of performance but we see that they follow the same structure found in the other curves suggesting that all curves are somehow converging towards an optimal solution.

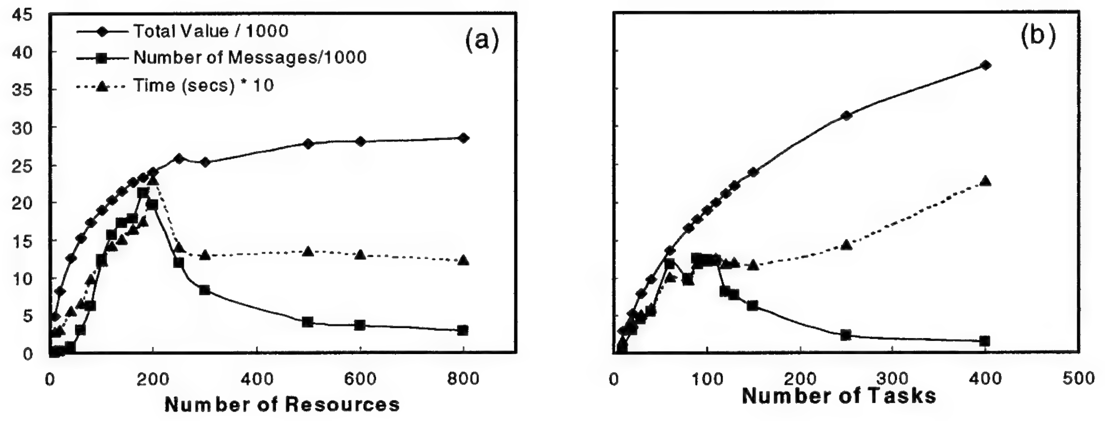


Figure 15: Easy-hard-easy phase-transition behavior of the total number of messages and computational time for the Marblesize scheme. (a) 100 tasks, (b) 100 resources

In Figure 15 we study the behavior of the total number of messages, execution time and total value of solutions found with Marblesize for different size of the problem. In Figure 15a, shows results for 100 tasks and different number of resources while in Figure 15b the results correspond to 100 resources and different number of tasks. In both curves we observe an easy-hard-easy phase-transition effect where the number of messages (and time) increases very drastically as the problem gets larger until it reaches a peak and after that drops down again. This property of Marblesize is due to the fact that unlikely to succeed tasks drop out of the competition very early in the process and do not waste any bidding messages. Since the distributions of task values and number of requirements per tasks are independent, tasks with large number of resources and low task value end up with marbles of relatively small size that makes them lose in all bids before entering competition. In Figure 15b, the execution time continues to rise slowly after the transition peak while the number of messages drops down and this is due to the fact that in the initial phase tasks need to evaluate alternative combination of resources before bidding and the total computational time of this operation increases with the number of tasks.

2.7 Work Related to MARBLES

What we are after are distributed negotiation schemes in which (1) domain experts can understand the decisions made by negotiation participants because they use a domain currency for making their trade-offs that the experts share, and that (2) can be "steered" in its collective real-time response, fault tolerance, and solution quality behavior by changing their relative desirability at run-time.

We list the most relevant non-market-inspired previous work on distributed resource allocation in the References section, but do not have the space here to discuss them at any length (they generally address neither (1) nor (2) above). Instead, we will use the remaining space to put the auction protocols we use for resource acquisition in the context of prior work. A negotiation protocol in our terms defines the types of messages that can be sent and how they can be strung together (the syntax of message exchange). In our terminology, this – together with the bidding strategies of requesters and the auctioning strategy of requesters – defines a "scheme" for market-based distributed resource allocation.

Walsh et al. (1998) outline the fundamental choices in this design space: (a) single-resource auctions, (b) combinatorial auctions, and (c) Vickery auctions. We view (b) combinatorial auctions as generally inapplicable to truly distributed assignment of resources. This is because they need a large number of messages to coordinate between themselves (as they cannot individually auction themselves but must bundle up with others to be bid on in combination). We cannot say with certainty that there may not be a space for them in real-time adaptive distributed resource allocation but we are not currently exploring this route. In (c) Vickery auctions, every resource requester has an incentive to report his true requirements to a centralized auction mechanism which can then make an optimal assignment of resources (solving an NP-complete problem) and report the assignments back to the requesters. This is obviously not an option for truly distributed resource allocation either, and we are not investigating this avenue further either.

This leaves (a) single-resource auctions, in which each resource can auction itself off to the highest-value task based solely on its local bid information. Our Marbles schemes are a subclass of single-resource auction. However, the distributed algorithms introduced for the altruistic task suicide phase further distinguish our Marbles schemes from work on competitive auctions. This task suicide phase is fundamental for the quick convergence of the Marbles schemes: by lowering resource prices it usually helps other tasks succeed.

2.8 MARBLES Conclusion

It is obviously far too early for us to make any claims on how far from "optimal" in any sense our currently implemented Marbles schemes are (be that in term of the quality of the solution, in terms of the number of messages needed, or any combination thereof). However we can conclude the following:

1. Marbles-type distributed collaborative negotiation schemes are an exciting and worthwhile research program for years to come; this is because there are many "optimal" solvers depending on how much the application domain values fault-tolerance, average response time, real-time response guarantee, and quality of the solution.

2. We are seeing "phase transitions" in our problems as is evidenced in Figure 15; to be precise, we are seeing Gauss-like curves for the amount of messages needed based on a varying number of resources for a fixed number of tasks. A Marbles scheme finds out

quickly that few tasks can be satisfied with the very few resources, as well as that nearly all tasks can be satisfied with the abundant resources, but uses substantially more computation if there are "just enough resources for most of the tasks with the right assignments". However, we currently have no way of predicting how much negotiation a given problem requires.

3. It seems that the Marbles schemes with good performance all seem to have the property of eliminating (apparently) losing tasks very early on.

4. As this is work in progress we have not compared our schemes against other distributed algorithms at great length. However, based on our performance comparisons of our best Marbles schemes to the well-known centralized Simulating Annealing strategy we believe that this family of market-inspired collaborative negotiation schemes is well-suited to the real-time distributed solution of resource allocation problems.

3 Robust UCAV Peer-to-Peer Knowledge Storage

In this section, we present a scalable Peer-to-Peer Resource Description Framework (RDF) repository, named RDFPeers. We chose RDF as the underlying knowledge representation for the UCAV peer-to-peer storage module as it is a new World Wide Web Consortium standard (a "recommendation") that we expect to be widely adopted and supported by third-party software. RDFPeers stores each RDF triple in a multi-attribute addressable network by applying globally known hash functions. Queries can be efficiently routed to the nodes that store matching triples. RDFPeers also supports users to selectively subscribe to RDF content. In RDFPeers, both the neighbors per node and the routing hops for triple insertion, most query resolution and triple subscription are logarithmic to the network size. Our experiments with real-world RDF data demonstrated that the triple-storing load among nodes in RDFPeers differs by less than an order of magnitude.

3.1 Introduction

Metadata is the foundation for the Semantic Web, and is also critical for Grid systems, [Deelman et al. 2002, Singh et al. 2003] , Peer-to-Peer systems [Nejdl 2002], as well as distributed agent systems like UCAV swarms. RDF (http://www.w3.org/RDF) metadata makes flexible statements about resources that are uniquely identified by URIs. RDF statements are machine-processable, and statements about the same resource can be distributed on the Web and made by different users. RDF schemata (http://www.w3.org/TR/rdf-schema) are extensible and evolvable over time by using a new base URI every time the schema is revised. The possibility to distribute RDF statements provides great flexibility for annotating resources.

However, distributed RDF documents on the Web are hard to discover. Putting an RDF document on a Web site does not mean that others can find it, much less issue structured queries against it. One approach is to crawl all possible Web pages and index all RDF documents in centralized search engines, ``RDF Google" if you wish, but this approach makes it difficult to keep the indexed metadata up to date. For example, it currently takes

Google many days to index a newly created Web page. Further, this approach has a large infrastructure footprint for the organization providing the querying service, and is a centralized approach on top of technologies (RDF, the Internet itself) that were intentionally designed for decentralized operations.

Also centralized RDF repositories are not well-suited for some semantic web applications in which data is not owned by any participant and each participant is responsible for supporting the community. When the community scales up, the data and query load will be so large that no participant will be able to afford the hosting cost. One example of such application is SciencePeers, a design for maintaining shared views of scientific fields in which the participants are peers both in the scientific sense (``peer review") and in the technical sense (``peer-to-peer technology"). In this design, each member takes responsibility for a proportional fraction of the disk storage, bandwidth, and computing cycles to support the community. Another example is Shared-HiKE, a collaborative hierarchical knowledge editor that lets users create, organize and share RDF data. In Shared-HiKE, each participant has her local hierachical knowledge and also shares the external knowledge from other participants.

Moreover, participants in the community want to quickly be notified of specific new content, that is, they have persistent queries expressing interest in certain people, products, or topics that are constantly serviced. For example, in addition to publishing and querying the shared views, each participant in SciencePeers can also subscribe to scientific topics of interest. In centralized RDF repositories without subscription support, this could only be accomplished by constantly issuing the queries of interest every few minutes, which will generate much unnecessary query load on the server. Also, it is difficult for centralized subscription schemes to scale up to a large number of subscribers. Thus, we argue that a distributed RDF infrastructure that can scale to Internet size and support RDF metadata subscription is useful or even necessary for many interesting semantic web applications, such as SciencePeers and Shared-HiKE.

One choice for non-centralized RDF repositories is Edutella [Nejdl 2002] that provides an RDF-based metadata infrastructure for P2P applications. It uses a Gnutella-like [Ripeanu et al. 2002] unstructured P2P network that has no centralized index or predictable location for RDF triples. Instead, RDF queries are flooded to the whole network and each node processes every query. Measurement studies [Saroiu et al. 2002, Sen and Wong 2002] show that Gnutella-like unstructured P2P networks do not scale well to a large number of nodes. This is because their flooding mechanism generates a large amount of unnecessary traffic and processing overhead on each node, unless a hop-count limit is set for queries - but then the queries cannot guarantee to find results, even if these results exist in the network. An Edutella successor [Nejdl et al. 2003] provides better scalability by introducing super-peers and schema-based routing; however, it requires up-front definition of schemas and designation of super peers.

This paper presents a scalable P2P RDF repository named RDFPeers that allows each node to store, query and subscribe to RDF statements. The nodes in RDFPeers self-organize into a cooperative structured P2P network based on randomly chosen node identifiers. When an RDF triple is inserted into the network, it will be stored at three places by applying a globally-known hash function to its subject, predicate, and object values. Both exact-match and range queries can be efficiently routed to those nodes where the matching triples are known to be stored if they exist. The subscriptions for RDF statements are also routed to and stored on those nodes. Therefore, the subscribers will be notified when matching triples are inserted into the network. We implemented a prototype of RDFPeers in Java and evaluated its preliminary performance and scalability in a 16-nodes cluster. We also measured the load balancing of real-world RDF data from the Open Directory Project by inserting it into a simulated RDFPeers network, and found that the load balances to less than an order of magnitude between the nodes when using a certain successor probing technique.

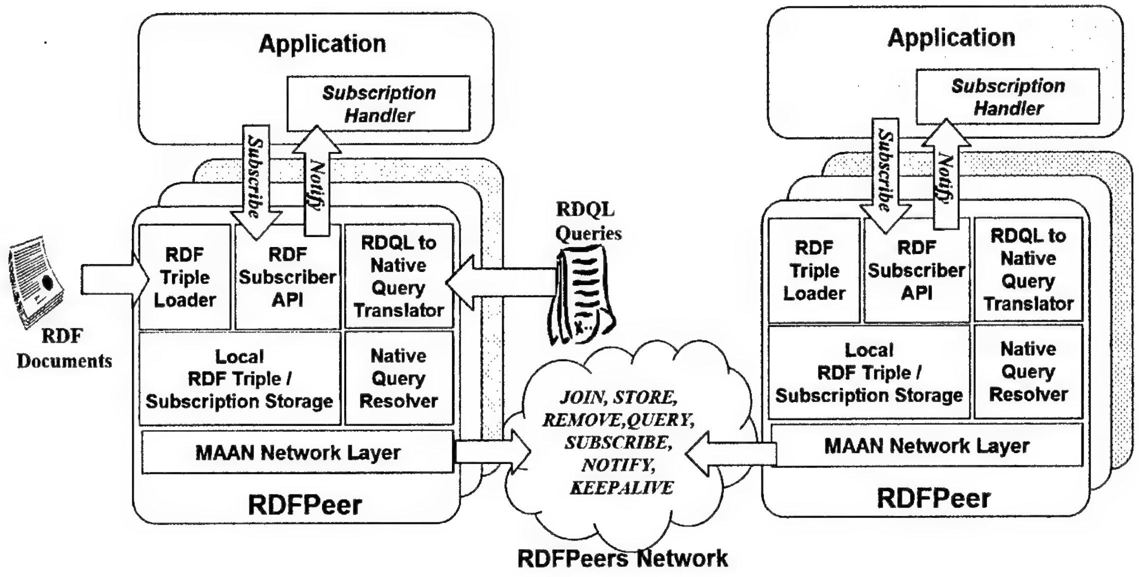


Figure 16: The Architecture of RDFPeers

3.2 RDFPeers Architecture

Our distributed RDF repository consists of many individual nodes called RDFPeers that self-organize into a multi-attribute addressable network (MAAN) [Cai et al. 2003]. MAAN extends Chord [Stoica et al. 2001] to efficiently answer multi-attribute and range queries. However, MAAN only supported predetermined attribute schemata with a fixed number of attributes. RDFPeers exploits MAAN as the underlying network layer and extends it with RDF-specific storage, retrieval, subscription and load balancing techniques. Figure 16 shows the architecture of RDFPeers. Each node in RDFPeers consists of six components: MAAN network layer, RDF triple loader, RDF Subscriber API, local RDF triple and subscription storage, native query resolver and RDQL-to-native-query translator.

The underlying MAAN protocol contains four classes of messages for (a) topology maintenance, (b) storage, (c) query, and (d) subscription. The topology maintenance messages are used for keeping the correct neighbor pointers and routing tables. It includes *JOIN*, *KEEPALIVE* and other Chord stabilizing messages. The *STORE* message inserts triples into the network and the *REMOVE* message deletes the triples from the network. The *QUERY* message visits the nodes where the triples in question are known to be stored, and returns the matched triples to the requesting node. The RDF triple loader reads an RDF document, parses it into RDF triples, and uses MAAN's *STORE* message to store the triples into the RDFPeers network. When an RDFPeer receives a *STORE* message, it stores the triples into its local RDF triple/subscription storage component such as a file or a relational database. The native query resolver parses native RDFPeers queries and uses MAAN's *QUERY* message to resolve them. There can be a multitude of higher-level query modules on top of the native query resolver that map higher-level user queries into RDFPeers' native queries, such as an RDQL to native query translator. Applications built on top of RDFPeers can also subscribe to RDF triples by calling the RDF subscriber API with a subscription handler. The RDFPeers node then sends a *SUBSCRIBE* message to the nodes that are responsible for storing matching triples. When the RDF triples that match the subscription are inserted into the network, the subscribing node will receive a *NOTIFY* message and notify the application to handle the triples with the subscription handler.

3.3 MAAN Overview

MAAN [Cai et al. 2003] uses the same one-dimensional modulo-2^m circular identifier space as Chord, where m is the number of bits in node identifiers and attribute hash values. Every node in MAAN is assigned a unique m-bit identifier, called the node ID, and all nodes self-organize into a ring topology based on their node IDs. The node ID can be chosen locally, for example by applying a hash function to the node's IP address and port number. In MAAN, bundles of related attribute-value pairs such as ``name: John, age: 27" are called ``resources", a term we will avoid in this paper because of its different meaning in RDF. Note that for RDFPeers' use of MAAN, a ``bundle of related attribute-value pairs" is always synonymous with ``an RDF triple".

Unlike Chord in which these bundles can only be stored and looked up by one unique key, they can be stored and looked up by any attribute value in MAAN. Chord uses SHA1 hashing [NIST 1995] to assign each key a unique m-bit identifier. MAAN uses the same hashing for string-valued attributes. However, for numeric attributes MAAN uses locality preserving hash functions to assign each attribute value an identifier in the m-bit space. Here, we refer to the hashing image of the key in Chord as well as to the hashing image of the attribute value in MAAN as ``the key" that is an identifier in the circular m-bit space. Suppose we have an attribute a, which has numeric values v which are elements of $[v_{min}, v_{max}]$. In RDFPeers, the only attributes that can have numeric values are the objects given that subjects and predicates are always non-numeric URIs in RDF. A simplistic locality preserving hash function we could use is
$H(v) = (v-v_{min}) * (2^m - 1)/(v_{max} - v_{min})$, where v is an element of $[v_{min}, v_{max}]$.

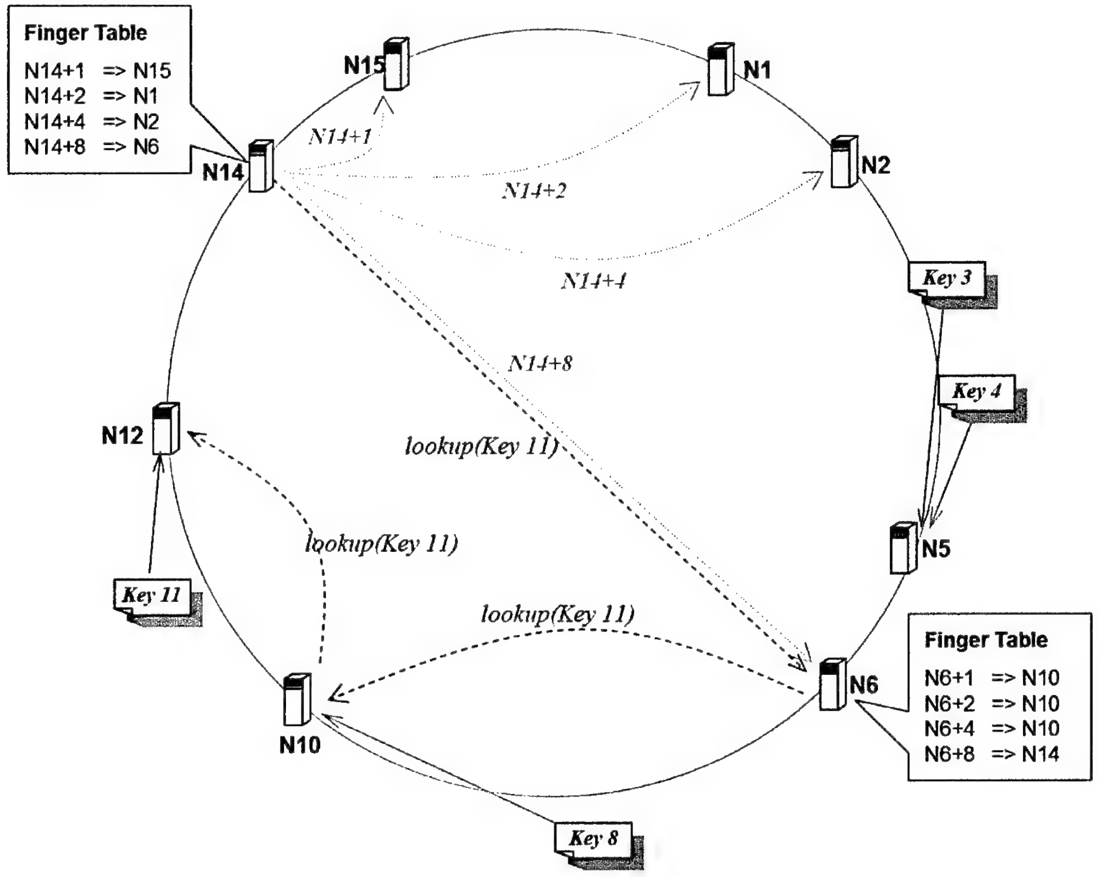


Figure 17: An 4-bit Chord network consisting of 8 nodes and 4 keys

In Chord, key *k* is assigned to the first node whose identifier is equal to or follows *k* in the identifier circle. This node is called the successor node of key *k*, denoted by *successor(k)*. Figure 17 shows an 8-node Chord network with 4-bit circular identifier space. Node *N5* has the node ID of 5 and stores the key 3 and key 4. Similar to Chord, each node in MAAN maintains two sets of neighbors, the *successor list* and the *finger table*. The nodes in the successor list immediately follow the node in the identifier space, while the nodes in the finger table are spaced exponentially around the identifier space. The finger table has at most *m* entries. The *i*-th entry in the table for the node with ID *n* contains the identity of the first node *s* that succeeds *n* by at least 2^{i-1} on the identifier circle, i.e. s = successor(n + 2^{i-1}), where 1<=i<=m and all arithmetic is modulo 2^m. The finger table contains more close nodes than far nodes at doubling distance. Thus each node only need to maintain the state for *O(log N)* neighbors for a network with *N* nodes. For example, the fingers of *N6* in are *N10* and *N14*. MAAN uses Chord's successor routing algorithm to forward a request of key *k* to its successor node. If a node *n* receives a request with key *k*, the node searches its successor list for the successor of *k* and forwards the request to it if possible. If it does not know the successor of *k*, it forwards the request to the node *j* whose identifier most immediately precedes *k* in its finger table. By repeating this process, the request gets closer and closer to the successor of *k*. For example, if *N14* in issues a lookup request for *Key11*, it sends the request to its finger *N6* that is the closest one to *Key11* in the identifier space. *N6* then forwards the request to *N10* that will forward it to *N12*. Since *N12* is the successor node of *Key11*, it looks up the resource corresponding to *Key11* locally and returns the result to *N14*. Since the fingers on each node are spaced exponentially around the identifier space, each hop from node *n* to the next node covers at least half the identifier space (clockwise) between *n* and *k*. The average number of hops for this routing is *O(log N)* for a network with *N* nodes.

MAAN stores each bundle of attribute-value pairs on the successor nodes of the keys for all its attribute values. Suppose each bundle has *M* pairs $<a_i, v_i>$ and $H_i(v)$ is the hash function for attribute a_i (Note that *M* is always 3 in RDFPeers, a_1 is always *subject*, a_2 is always *predicate*, and a_3 is always *object*.) Each bundle of attribute-value pairs will be stored at node $n_i = successor(H(v_i))$ for each attribute value v_i, where $1<=i<=M$. A *STORE* message for attribute value v_i is routed to its successor node using the above successor routing algorithm. *M* nodes store the same bundle consisting of attribute-value pairs, each by keying on a different attribute. Thus, the routing hops for storing a bundle of attribute-value pairs is *O(M log N)* for bundles with *M* attributes.

Since numeric attribute values in MAAN are mapped to the *m*-bit identifier space using locality preserving hash function *H*, numerically close values for the same attribute are stored on nearby nodes. Given a range query *[l,u]* where *l* and *u* are the lower bound and upper bound respectively, nodes that contain attribute value *v* element of *[l,u]* must have an identifier equal to or larger than *successor(H(l))* and equal to or less than *successor(H(l))*.

Suppose node *n* wants to search for bundles with attribute value *v* element of *[l,u]* for attribute *a*. Node *n* composes a *QUERY* message and uses the successor routing algorithm to route it to node n_l, the successor of *H(l)*. The query message has parameters *k*, *a*, *R*, and *X*. *k* is the key used for successor routing, initially $k = H(l)$. *a* is the name of the attribute we are interested in, *R* is the desired query range *[l,u]* and *X* is the list of bundles of attribute-value pairs discovered in the range. Initially, *X* is empty. When node n_l receives the query message, it searches its local sets and appends those sets that satisfy the range query for attribute *a* to *X* in the message. Then it checks whether it is the successor of *H(u)* also. If true, it sends back the query result in *X* to the requesting node *n*. Otherwise, it forwards the query message to its immediate successor n_i. Node n_i repeats this process until the message reaches node n_u, the successor of *H(u)*. Thus, routing the query message to node n_l via successor routing takes *O(log N)* hops for *N* nodes. The next sequential forwarding from n_l to n_u takes *O(k)*, where *K* is the number of nodes between n_l and n_u. So there are total O(log N + K) routing hops to resolve a range query for one attribute. Given that the nodes are uniformly distributed in the *m*-bit identifier space, *K* is $N * s$ where *s* is the selectivity of the range query and $s = (l - u)/v_{max} - v_{min})$.

MAAN supports multi-attribute and range queries using a single-attribute-dominated query resolution approach. Suppose *X* are the bundles of attribute-value pairs satisfying all sub-queries, and X_i are the bundles satisfying the sub-query on attribute a_i, where $1<=i<=M$. So we have *X* as the intersection of the X_i, and each X_i is a superset of *X*. This query resolution approach first computes a X_k that satisfies one sub-query on attribute a_k,. Then it applies the sub-queries for other attributes on these candidate bundles and computes the intersection X that satisfies all sub-queries. Here, we call attribute a_k the dominant attribute. In order to reduce the number of the candidate sets that do not satisfy other sub-queries, we carry all other sub-queries in the *QUERY* message, and use them to filter out the unqualified bundles of attribute-value pairs locally at the nodes visited. Since this approach only needs to do one iteration around the Chord identifier space for the dominant attribute a_k, it takes $O(\log N + N * s_k)$ routing hops to resolve the query, where s_k is the selectivity of the sub-query on attribute a_k. We can further minimize the routing hops by choosing the attribute with minimum selectivity as the dominant attribute, presuming, of course, that the selectivity is known in advance; in that case, the routing hops will be $O(\log N + N * s_{min})$, where s_{min} is the minimum range selectivity for all attributes in the query.

Although the simplistic locality preserving hash function above keeps the locality of attribute values it does not necessarily produce uniform distributions of hashing values if the distribution of attribute values is not uniform. Consequently, the load balancing of resource entries can be poor across the nodes. To address this problem, we proposed a uniform locality preserving hashing function in MAAN that always produces uniform distribution of hashing values if the distribution function of input attribute values is continuous and if the distribution is known in advance. The former condition is satisfied for many common distributions, such as Gaussian, Pareto, and Exponential distributions. Suppose attribute value v conforms to a certain distribution with continuous and monotonically increasing distribution function $D(v)$ and possibility function $P(v) = dD(v)/dv$, and v is an element of $[v_{min}, v_{max}]$. We can design a uniform locality preserving hashing function $H(v)$ as follows: $H(v) = D(v) * (2^m - 1)$. This load balance mechanism assumes that we know the distribution functions of attribute values in advance. However, this prerequisite is not always true for many semantic web applications. Section 3.13 discusses our approach to dynamically balance load among nodes by probing the load on multiple successors when new nodes join.

3.4 Storing RDF Triples

RDF documents are composed of a set of RDF triples. Each triple is in the form of *subject, predicate, object.* The *subject* is the resource about which the statement was made. The *predicate* is a resource representing the specific property in the statement. The *object* is the property value of the predicate in the statement. The object is either a resource or a literal; a resource is identified by a URI; literals are either plain or typed and have the lexical form of a unicode string. Plain literals have a lexical form and optionally a language tag, while typed literals have a lexical form and a datatype URI. The following triples show three different types of objects, resource, plain literal, and typed literal, respectively.

```
@prefix info: <http://www.isi.edu/2003/11/info#>  .
@prefix dc:   <http://purl.org/dc/elements/1.1/>  .
@prefix foaf: <http://xmlns.com/foaf/0.1/>  .
```

```
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

<info:rdfpeers> <dc:creator> <info:mincai>  .
<info:mincai>   <foaf:name>  "Min Cai" .
<info:mincai>   <foaf:age>   "28"^^<xsd:integer> .
```

In order to support efficient queries on distributed RDF triples, we exploit the overlay structure of MAAN to build a distributed index for these triples. In the *STORE* message one of the three attribute values is designated as the destination of the routing, and we store each triple three times, once each based on its subject, predicate, and object. Each triple will be stored at the successor node of the hash key of the value of the routing key attribute-value pair. Since the value of attribute ``subject" and ``predicate" must be a URI that is a string, we apply the SHA1 hash function to mapping the subject value and predicate value to the *m*-bit identifier space in MAAN. However, the values of attribute ``object" can be URIs, plain literals or typed literals. Both URIs and plain literals are strings and we apply SHA1 hashing on them. The typed literal can be either string types or numeric types, such as an enumeration type or a positive integer respectively. As discussed above, we apply SHA1 hashing on string-typed literals and locality preserving hashing on numeric literals. For example, to store the first triple above by subject, RDFPeers would send the following message in which the first attribute-value pair *(``subject", info:rdfpeers)* is the routing key pair, and *key* is the SHA1 hash value of the subject value.

```
STORE {key, {("subject", <info:rdfpeers>),
             ("predicate", <dc:creator>),
             ("object", <info:mincai>)}}
where key=SHA1Hash("<info:rdfpeers>")
```

This triple will be stored at the node that is the successor node of *key*. Figure 18 shows how the three triples above are stored into an example RDFPeers network. It also shows the finger tables of example nodes *N6* and *N14* for illustration.

Most semantic web applications prefer high availability to strong consistency in the face of network partitions. RDFPeers provides relaxed consistency by leveraging soft state updates [Clark 1988, Cai et al. 2004]. Each triple has an expiration time, and the node that inserts the triple needs to renew the triple before it expires. If the nodes that store the triple do not receive any renewals, the triple will be removed from their local storage. With soft state updates, RDFPeers provides best effort consistency for triples indexed at three places.

To apply locality preserving hashing on numeric literals of RDF triples, we need to know their minimal and maximal values. We can leverage the datatype information of the predicates provided by the RDF Schema definition. For example, the following schema defines that the object values of triples whose predicate is <foaf:age> are instances of *<info:AgeType>*.

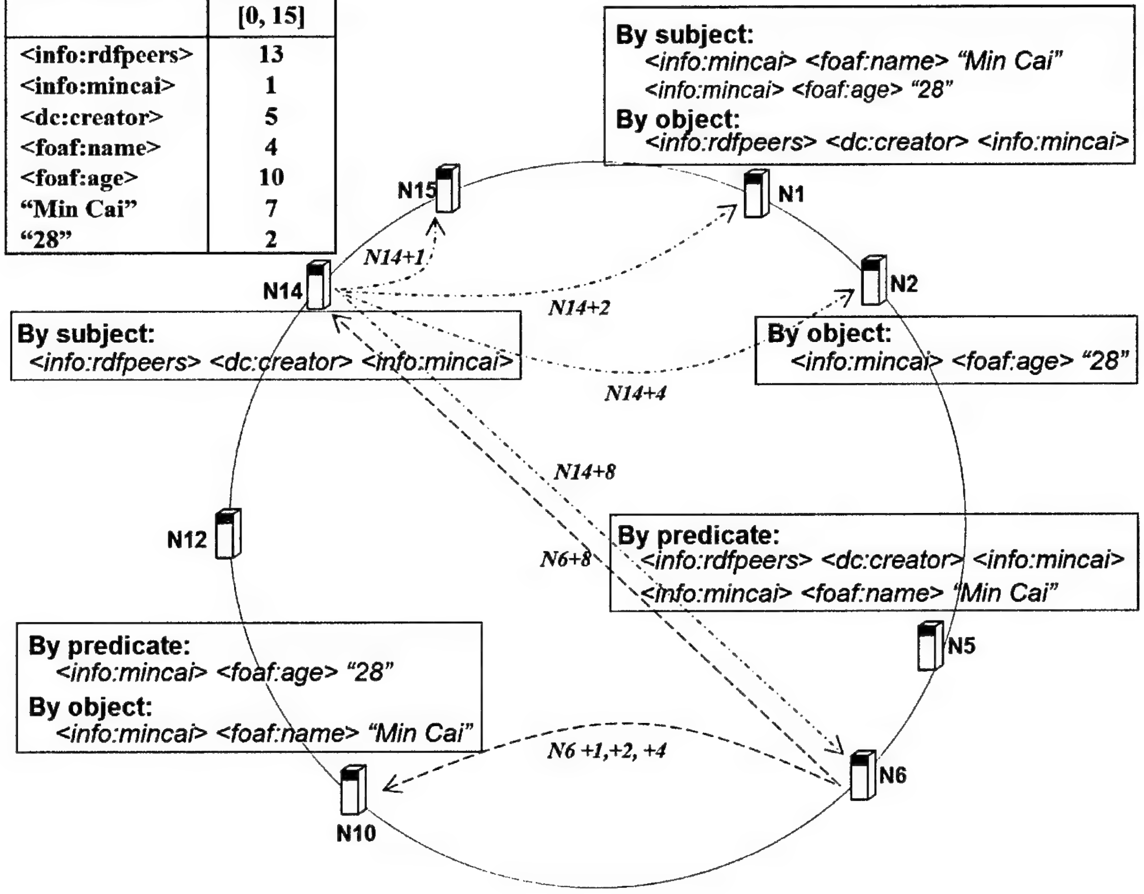
```
<foaf:age> <rdfs:range> <info:AgeType> .

<xsd:simpleType name= "info:AgeType">
    <xsd:restriction base="xsd:integer">
```

```
        <xsd:minInclusive value="0"/>
        <xsd:maxInclusive value="250"/>
    </xsd:restriction>
</xsd:simpleType>
```

Thus the numeric literals have the minimal value 0 and maximal value 250 as defined by the XML schema of *<info:AgeType >*. However, this approach assumes that the datatypes of predicates are fixed and will not change in the future. We solve this problem by only allowing a schema to envolve to a new version with a different namespace rather than change the old version; and applications are responsible for updating the RDF statements with new schemas, which is always necessary because of our soft state scheme.

URI / Literal	Hash Value in [0, 15]
<info:rdfpeers>	13
<info:mincai>	1
<dc:creator>	5
<foaf:name>	4
<foaf:age>	10
"Min Cai"	7
"28"	2



By subject:
<info:mincai> <foaf:name> "Min Cai"
<info:mincai> <foaf:age> "28"
By object:
<info:rdfpeers> <dc:creator> <info:mincai>
N15
N1
N14+1
N14
N2
N14+2
By subject:
<info:rdfpeers> <dc:creator> <info:mincai>
By object:
<info:mincai> <foaf:age> "28"
N14+4
N14+8
N12
N6+8
By predicate:
<info:rdfpeers> <dc:creator> <info:mincai>
<info:mincai> <foaf:name> "Min Cai"
N5
By predicate:
<info:mincai> <foaf:age> "28"
By object:
<info:mincai> <foaf:name> "Min Cai"
N6 +1,+2, +4
N6
N10

Figure 18: Storing three triples into an RDFPeers network of eight nodes in an example 4-bit identifier space that could hold up to 16 nodes. (In reality a much larger identifier space is used, such as 128 bits.)

Since nodes might fail and network connections might break, triples stored on their corresponding successor nodes are replicated on their neighbors in the Chord network. This can be done by setting the parameter *Replica_Factor* in MAAN. Whenever a node receives a triple storing request, it will not only store the triple locally but also store it to as many of its immediate successors as the above parameter dictates. If any node fails or its connection breaks, its immediate successor and predecessor will detect it by checking the *KEEPALIVE* messages. If the node does not come back to life after a time-out period, nodes will repair the ring structure using the Chord stabilization algorithm. After stabilization, the immediate successor node of the failed node will restore its replicas to its new predecessor.

3.5 Native Queries in RDFPeers

Based on the above triple-storing scheme, we define a set of native queries that can be efficiently resolved via MAAN's multi-attribute range queries. These native queries include atomic triple queries, disjunctive and range queries, and conjunctive multi-predicate queries.

3.5.1 Atomic Triple Queries

An atomic query query is a triple pattern in which the subject, predicate, or object can each either be a variable or an exact value. The eight resulting possible queries are shown in Figure 19.

Q1 is the most general and most expensive query that matches all triples. Since there is no restriction whatsoever on this triple pattern, we have to propagate this query to all nodes, which takes *O(N)* routing hops for a network with *N* nodes.

We can use MAAN's routing algorithm to resolve queries *Q2* through *Q8* since we store each triple three times based on its subject, predicate, and object hash values. In these seven query patterns, there is always at least one value that is a constant, and we resolve the query by routing it to the node responsible for storing that constant, that node then matches these triples against the pattern locally and returns them to the requesting node. For example, in Figure 18 if node *N6* asks the native query (*<info:mincai>*, *<foaf:name>*, *?name*), we hash on *info:mincai* and get the hash value ``1". Then *N6* routes it to the corresponding node *N1* (via *N14*). *N1* filters triples locally using this pattern, and sends back the matched triple *<info:mincai>*, *<foaf:name>*, *``Min Cai"* to N6 (via N5).

No.	Query Pattern	Cost	Query Semantics
Q1	$(?s, ?p, ?o)$	O(N)	find all possible triples
Q2	$(?s, ?p, o_i)$	log N	given object o_i of any predicate, find the subjects and predicates of matching triples
Q3	$(?s, p_i, ?o)$	log N	given predicate p_i, find the subjects and objects of the triples having this predicate

Q4	$(?s, p_i, o_i)$	log N	given object o_i of predicate p_i, find the subjects of matching triples
Q5	$(s_i, ?p, ?o)$	log N	given subject s_i, find all predicates and objects of the resource identified by s_i
Q6	$(s_i, ?p, o_i)$	log N	given subject s_i, find its predicate that has object o_i
Q7	$(s_i, p_i, ?o)$	log N	given subject s_i, find its object of predicate p_i
Q8	(s_i, p_i, o_i)	log N	return this triple if it exists otherwise return nothing

Figure 19: The eight possible atomic triple queries for exact matches. The cost is measured in the number of routing hops needed to resolve each query

3.5.2 Disjunctive and Range Queries

RDFPeers' native queries support constraints on variables in the triple patterns. Q9 extends the above atomic triple queries with a constraint list that limits the domain of variables.

```
Q9               ::= TriplePattern 'AND' ConstraintList
TriplePattern    ::= Q1|Q2|Q3|Q4|Q5|Q6|Q7
ConstraintList   ::= OrExpression ('&&' OrExpression)*
OrExpression     ::= Expression ('||' Expression)*
Expression       ::= Variable (NumericExpression
                            | StringExpression)+
NumericExpression ::=('>'|'<'|'='|'!='|'<='|'>=')
                     NumericLiteral
StringExpression ::= ('='|'!=')Literal
Literal          ::= PlainLiteral|URI|NumericLiteral
```

Variables can be either string-valued or numeric. Constraints can limit the domain of string values by enumerating a set of either allowed or forbidden constants. Numeric variables can additionally be limited to a set of disjunctive ranges.

```
(a) (?s, dc:creator, ?c) AND ?c=''Tom'' || ?c=''John''
(b) (?s, foaf:age, ?age) AND ?age > 10 && ?age < 20
```

As discussed in Section 3.3 , MAAN can efficiently resolve range queries by using locality preserving hashing. In addition to specifying a single range, Q9 can also specify a set of disjunctive ranges for attribute values. For example, a user can submit a range query for variable *?x* where *?x* is an element of the union from *i*=1 through *d* of $[l_i, u_i]$.

Obviously, this kind of disjunctive range query could simply be resolved by issuing one query for each contiguous range and by then computing the union of the results. For a query with d disjunctive ranges, this takes $d * O(\log N + N * s)$, where s is the aggregate selectivity of the d ranges. So the number of hops in the worst case increases linearly with d and is not bounded by N. We can optimize this by using a range ordering algorithm that sorts these disjunctive query ranges in ascending order. Given a list of disjunctive ranges in ascending order, $[l_i, u_i]$, $1 <= i <= d$ where $l_i <= l_j$ and $u_i <= u_j$ iff $i <= j$, the query request will be first routed to node n_{l1}, the successor node of $H(l_1)$ that is the key corresponding to the lower bound of the first range. Node n_{l1} then sequentially forwards the query to the successor node of the upper bound $H(u_1)$ if it itself is not the successor node of $H(u_1)$. Then node n_{u1} uses successor routing to forward the query to node n_{l2}, the successor node corresponding to the lower bound of the next range $[l_2, u_2]$, which in turn forwards the query to the successor node of $H(u_2)$. This process will be repeated until the query reaches the successor node of $H(u_d)$. This optimized algorithm exploits the locality of numeric MAAN data on the Chord ring and the ascending order of the ranges, reduces the number of routing hops, especially for cases where d is large, and bounds the routing hops to N. Disjunctive exact-match queries such as *?c element-of {Tom, John}* present a special case of the above disjunctive range queries where both the lower bound and upper bound of the range are equal to the exact-match value, and we use the same algorithm to resolve them.

3.6 Conjunctive Multi-Predicate Queries

In addition to atomic triple queries and disjunctive range queries, RDFPeers handles conjunctive multi-predicate queries that describe a non-leaf node in the RDF graph by specifying a list of edges for this node. They are expressed as a conjunction of atomic triple queries or disjunctive range queries for the same subject variable. Q10 consists of a conjunction of sub-queries where all subject variables must be the same.

```
Q10            := TriplePatterns 'AND' ConstraintList
TriplePatterns := (Q3|Q4|Q9)+
```

In Q10, we restrict the sub-query Q9 to be the Q3-style triple pattern with constraints on the object variable. Thus Q10 describes a subject variable with a list of restricting *predicate, object* or *predicate, object-range* pairs.

```
(?x, <rdf:type>, <foaf:Person>)
(?x, <foaf:name>, "John")
(?x, <foaf:age>, ?age) AND ?age > 35
```

To efficiently resolve these conjunctive multi-predicate queries, we use a recursive query resolution algorithm that searches candidate subjects on each predicate recursively and intersects the candidate subjects inside the network, before returning the query results to the query originator. The query request takes the parameters q, R, C, and I, where q is the currently active sub-query, R is a list of remaining sub-queries, C is a set of candidate subjects matching current active sub-query, and *I* is a set of intersected subjects matching all resolved sub-queries. Initially, *q* is the first sub-query in this multi-predicate query, *R* contains all sub-queries except *q*, *C* is empty and *I* is the whole set. Suppose the sub-query *q* for predicate p_i is $v_{li} <= o_i <= v_{ui}$, where v_{li} and v_{ui} are the lower bound and upper bound of the query range for the object variable o_i, respectively. When node *n* wants to issue a query request, it first routes the request to node $n_{li} = successor(H(v_{li}))$. The node n_{li} receives the request, searches its local triples corresponding to predicate p_i, appends the subjects matching sub-query *q* to *C*, forwards this request to its immediate successor n_{si} unless it is already the $successor(H(v_{ui}))$. Node n_{si} repeats this process until the query request reaches node $n_{ui} = successor(H(v_{ui}))$. When node n_{ui} receives the request, it also searches locally for the subjects matching sub-query *q* and appends them to *C*. It then intersects set *I* with set *C*, and pops the first sub-query in *R* to *q*. If *R* or *I* is empty, it sends the query response back with the subjects in *I* as the result; otherwise, it resolves sub-query *q*. This process will be repeated until no sub-queries remain or *I* is empty.

This recursive algorithm takes $O(\sum_{i=1}^{k} (\log N + N * s_i))$ routing hops in the worst case, where *k* is the number of sub-queries and s_i is the selectivity of the sub-query on predicate p_i. However, it intersects the query results on different predicates in the network and will terminate the query process before resolving the query on all predicates if there are no matches left, i.e. *I* is empty. Thus, we can further reduce the average number of expected routing hops by sorting the sub-queries in ascending order of selectivity presuming the selectivity can be estimated in advance. For example, in the above three-predicate query, the sub-query on *rdf:type* might match many subjects, while *foaf:age* matches far fewer and *foaf:name* matches only a handful. After sorting the sub-queries, we resolve *foaf:name* first, then *rdf:age*, and finally *rdf:type.*

3.7 Resolving RDQL Queries

RDQL [Miller at al. 2002] is a query language for RDF proposed by the developers of the popular Jena Java RDF toolkit [McBride 2001]. RDQL operates at the RDF triple level, without taking RDF Schema information into account (like RQL [Karvounarakis 2002] does) and without providing inferencing capabilities. As such, it is the type of low-level RDF query language that we want RDFPeers to support well. It is our intuition that it is possible to translate all RDQL queries into combinations of the native RDFPeers queries above; however, we have not yet written such a translator and it may be inefficient for some queries, especially for joins. This section informally describes how the example RDQL queries from the Jena tutorial (http://www.hpl.hp.com/semweb/doc/tutorial/RDQL) would be resolved.

```
(1) SELECT ?x WHERE  (?x, <vcard:FN>, "John Smith")

(2) SELECT ?x, ?fname WHERE  (?x, <vcard:FN>, ?fname)

(3) SELECT ?givenName
    WHERE  (?y, <vcard:Family>, "Smith"),
          (?y, <vcard:Given>, ?givenName)

(4) SELECT ?resource
    WHERE (?resource, <inf:age>, ?age) AND ?age>=24

(5) SELECT ?resource, ?givenName
    WHERE (?resource, <vcard:N>, ?z),
          (?z, <vcard:Given>, ?givenName)

(6) SELECT ?resource, ?familyName
    WHERE (?resource, <inf:age>, ?age),
          (?resource, <vcard:N>, ?y),
          (?y, <vcard:Family>, ?familyName) AND ?age>=24
```

Query (1) translates directly into Q4, so that it can be resolved in *log N* routing hops in a network of *N* nodes. Similarly, query (2) translates directly into Q3, taking *log N* hops. To resolve query (3) , we first issue a Q4-style query and then use its query result as constraint to issue a Q9-style disjunctive query with Q3-style triple patterns. Since all the predicate values in the two triple patterns are known, these two native queries can be resolved in 2 * *log N* hops. Query (4) is a typical Q9-style range query with the constraint on the object value. Since its predicate value is known, we can route the query to the node that stores the triples with predicate *inf:age* in *log N* hops.

Our native queries do not include join operations, so that we decompose join queries into multiple native queries. Query (5) can be resolved via two Q3-style queries, and by then joining the first triple set's object with the second triple's subject, 2 * *log N* routing hops. (However, note that these two Q3-style queries might generate large-size messages if the predicates *vcard:N* or *vcard:Given* are popular.). Query (6) can be resolved by first issuing the same query as for the previous RDQL example for the first triple pattern. Then we use the query result as a constraint for variable *?resource* and resolve the second triple pattern as a Q9-style disjunctive range query. Finally, we use the second query result as a constraint for variable *?y* and again resolve the third triple as a Q9-style query, which in the aggregate takes 3 * *log N* hops.

3.8 RDF Subscription and Notification

3.8.1 Subscribing to Atomic Queries

RDFPeers also already implements subscriptions for atomic queries in which at least one of the triple's values is restricted to a constant. Our basic scheme for subscriptions to these queries is that the subscription request is routed to the same node that is responsible for storing the triple with that value in that position. Thus, the subscription request for (*?person*, *?predicate*, ``*Min Cai*'') would be routed to node *N10* in the example of. If there are multiple constants in the triple pattern, absent a-priori knowledge of the frequency distribution, we heuristically bias to first use the subject, then the object, then the predicate to route subscriptions, based on our experience for which positions are most likely to have ``overly popular'' values (see Section 3.12 ). Thus, the subscription request for (*?person*, *<foaf:age* >, ``*28"*) would be routed to node *N2* in our running example.

Each node keeps a local list of subscriptions, which consist of (1) a triple pattern, (2) a requested notification frequency, (3) the requested expiration date of the subscription, and (4) the node identifier of the subscriber. Each node internally maintains hash-based access into this subscription list where the key is a position-constant pair. When a node stores or removes a triple, it will also locally evaluate the matching subscription queries and (immediately or after collecting several such matches) notify the subscribing node of the matched triples. How often such notification messages are sent is dictated by the larger duration of (a) the requested notification frequency, and (b) a minimum interval between updates that the subscription-hosting node may impose. Given that we want both data and subscriptions to survive the sudden death of any node, we replicate the subscription list to the next replication factor nodes in the identifier space, just as we do for the triples themselves. The repair protocol for subscription data is identical to the repair protocol for triple data described in Section 3.4 . Conversely, it is possible that subscriptions persist after the subscribing node has quit the network. We deal with this issue via a maximum subscription duration parameter. Each node will periodically purge its subscriptions list from older entries, unless a subscribing node re-issued the subscription request more recently, in which case it will reset the age of the subscription to the latest request date.

3.8.2 Subscribing to Disjunctive and Range Queries

In disjunctive and range queries, the object is restricted to be within one or more disjunctive enumeration domains or numeric ranges. The basic subscription scheme for disjunctive and range queries is similar to the one for constant queries, but the subscription request is stored by all nodes that fall within the hashed identifiers of the minimum and maximum range value. The routing of the subscription request is identical to that for range queries described in Section 3.5.2 , taking *O(log N + N * s)* routing hops where *s* is the selectivity of the range query. For performance reasons, large-selectivity range query subscriptions are undesirable because a large number of matches would be sent. In practice, a query for common integer values independent of a target predicate such as (a) likely makes little sense. However, subscriptions for a narrow date range as in (b) or historical date ranges (``the 12th century") independent of predicate seems to be of practical value.

```
(a) ( ?, ?, ?age) AND ?age > 0
(b) (?, ?, 2004-04-13T00:00:00Z<=?<2004-04-16T00:00:00Z)
```

Conceivably, the network could reject range subscription requests that span more than maximum range subscription selectivity nodes (say, 20). For example, this could be done by having the node to which the minimum value hashes to - and which thus is the first node in the series of nodes that would add it to its subscription list - compute the estimated number of nodes that would be involved (selectivity in percentage of the identifier space times estimated number of participating nodes, the latter estimated from the size of the node's finger table), and reject it if it exceeds that threshold. This technique would not prevent a node from inserting a range request that did not exceed the threshold at insertion time but does exceed it later because of network growth, but it would prevent the subscription from being re-inserted as-is into the larger network once the original subscription expires. At present, we have not implemented such a rejection mechanism for overly broad range subscriptions.

3.8.3 Subscribing to Conjunctive Multi-Predicate Queries

These conjunctive multi-predicate queries look for subjects that match multiple constant (or constant range) predicate-object pairs. We have not implemented these subscriptions at time of writing. A possible scheme could initially route the subscription to the node corresponding to the first clause, which would remove and store just this first clause, and would also store the hash value of the next clause (not the identifier of the node that it currently maps to, given that the clause will move as nodes appear and disappear). Another node will then store the next clause, and route the remaining clauses towards the hash value of the next closest clause, and so on. The node storing the last clause will store the node identifier of the node that issued the original subscription request. Then, whenever the first clause match a new triple, the matching triples will be forwarded to the second node in the chain. The second and subsequent nodes will only further forward those triples that also match their local filtering criterion. There is a complication to this scheme if range queries are involved. In these cases, in the subscription registration phase one would always propagate the subscription request to the next nearest node involved.

3.8.4 Unsupported Subscription Types

We do not support subscriptions to the following types of queries.

```
1.  (?, ?, ?)
2.  (…) AND ((…) OR (…))
3a. (?x, <a>, ?) AND (?x, <b>, ?)
3b. (?, ?, ?x) AND (?x, ?, ?)
```

The first type is inherently not scalable to large networks, and we do not intend to ever support it. The second type consists of a combination of disjunctive and conjunctive sub-queries. It is our intuition that this type of subscription could be supported by chaining the techniques explained above within the network, with the leaves of the query parse tree as the starting points. The third type consists of joins. It is our intuition that those joins could be supported for which (a) one of the conjunctive clauses has a constant value and for which (b) this clause by itself matches only a moderate number of triples (example 3a). With the design of RDFPeers as described, it may not be possible to efficiently resolve joins for which each clause by itself leads to an overwhelming number of triples while the join between them leads to few (example 3b). However, the following slight variation may allow RDFPeers to handle those as well. Instead of prefixing URIs and values with ``subject:", ``predicate:", and ``object:" before applying SHA-1 hashing, as we do now for a slight load balancing gain, we could instead not add that prefix before hashing. In that case, the same URI hashes to the same node regardless of position. At a high cost of *O(N)* routing hops for first broadcasting the subscription to all nodes, each node can then locally search for a match when it stores a new triple and notify the subscriber.

3.8.5 Extension To Support Highly Skewed Subscription Patterns

In a real-world P2P application, e.g. of scientists subscribing to each others' Weblog-like communications, it is likely that the vast majority of scientists will have few if any subscribers while a handful will attract nearly everybody. In the latter case, analogous to IP multicasting for Internet video streaming, we want to avoid having to originate a number of messages proportional to the number of subscribers from the subscription-handling node, but rather want to construct something more akin to a real-life ``phone tree". We propose the following possible extension: If a node ends up with multiple identical subscription queries by different subscribers, such as (<mailto:famous@ivy-league.edu>, ?, ?), it will internally combine them into a single entry in its subscription list, with multiple addresses to be notified. It will then designate the node half-way across from it in identifier space as a replicating node for the subscriptions if the number of subscribers exceeds a certain threshold, then one a quarter-way across if an-other threshold is exceeded, and so on, adding up to *log N* ``repeater nodes" analogous in structure to its finger table. In this case, it will send the repeater nodes those subscriber identifiers that fall into their responsibility. The repeater nodes can then themselves set up repeater nodes, and so on. In the aggregate, this will take *O(N)* notification messages if all *N* nodes subscribe (the natural lower bound), and additionally leads to the busiest nodes having to send no more than *O(log N)* notification messages rather than the *O(N)* that need to be sent from the subscription-handling node in the naive approach.

3.9 Implementation and Evaluation

We implemented a prototype of RDFPeers in Java that extends our previous MAAN implementation. RDFPeers is implemented as a Java library and exposes the following API to applications:

```
interface RDFPeersJavaApi {
    public void store(Triple t);
    public void remove(Triple t);
    public Iterator query(TriplePattern p);
    public void subscribe(
        TriplePattern p,
        long durationOfSubscriptionInMs,
        long minimumTimeBetweenNotificationsInMs,
        SubscriptionHandler h);
}

interface SubscriptionHandler {
    public void notify(Iterator added, Iterator deleted);
}
```

The first two methods store and remove RDF triples from the P2P network. The query method lets you retrieve triples from the network by specifying a triple pattern that can restrict values to be constants or numeric ranges. The subscribe method lets you watch for RDF content changes by passing a triple pattern to watch for, how long you would like this subscription to last (in milliseconds), how frequently you want to be notified, and a call-back object; you will then be called back periodically with lists of added and deleted triples that matched. All of these four methods throw a variety of exceptions not further described here, such as ones for broken connections and response time-outs.

We already measured the performance of MAAN on a real-world network of up to 128 nodes in a previous paper [Cai et al. 2003]. We measured the number of neighbors per node against the network size. Similar to Chord, the number of neighbors at each node increases logarithmically with the network size, so that the node state in MAAN scales well to a large number of nodes. We also measured the number of routing hops against the network size for both exact-match queries and for range queries. The experiment results showed that for exact-match queries, the number of routing hops in the worst case is *O(log N)* and the average routing hops is *log N/2*. However, for range queries whose selectivity s_i > *epsilon%*, meaning that they select more than one node, the routing hops increase linearly with network size. This is optimal in the sense that s_i of total *N* nodes have to be visited by the search queries presuming we want to evenly balance the load to the nodes.

3.10 Routing Hops to Resolve Native Queries

The number of routing hops taken to resolve a query is the dominant performance metric for P2P systems. Figure 20 shows our simulation result for atomic triple patterns from 1 node to 8192 nodes on a logarithmic scale, which matches our theoretical analysis.

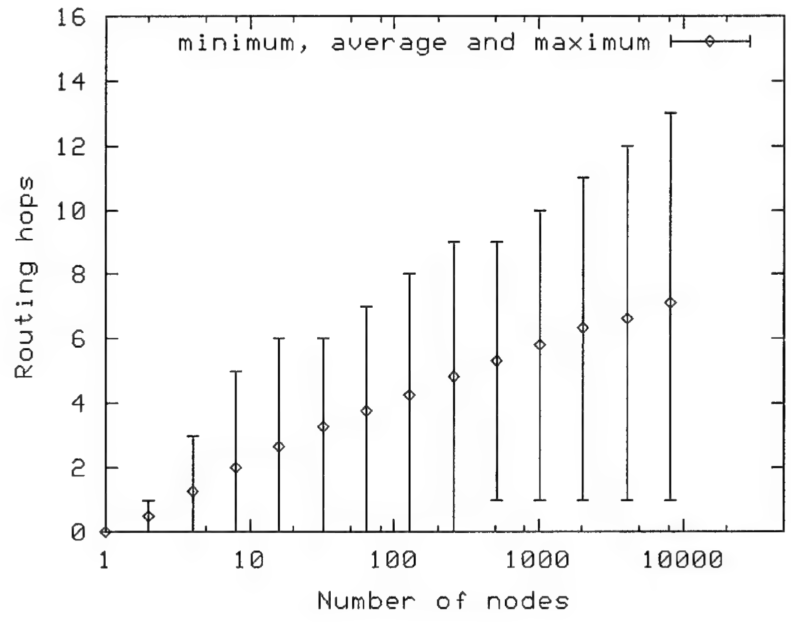


Figure 20: number of routing hops to resolve atomic triple patterns Q2 through Q8

We also compared two disjunctive range query resolution algorithms: the simple algorithm vs. range ordering algorithm. Figure 21 shows the simulation result for up to 1000 disjunctive exact-match values (s_i = *epsilon%*) in a network with 1000 nodes.

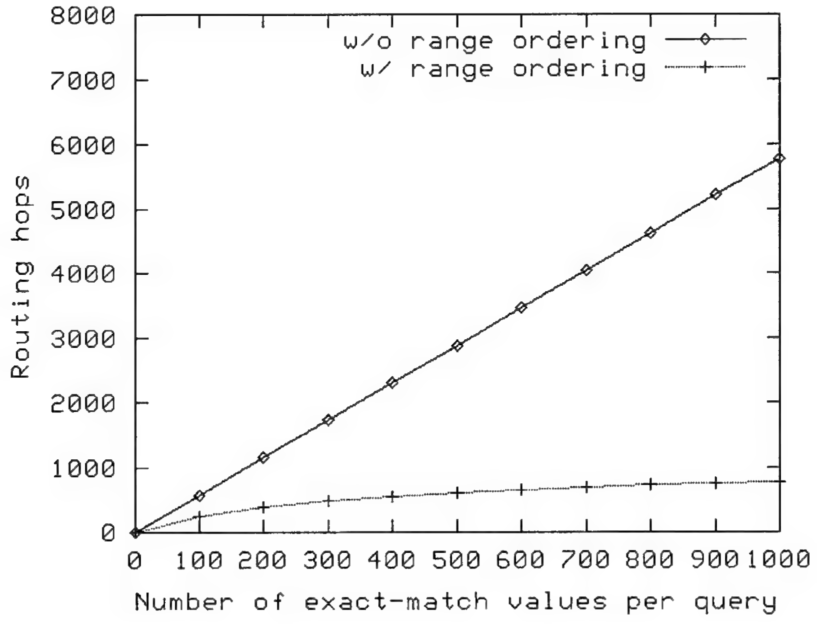


Figure 21: The number of routing hops to resolve disjunctive exact-match queries in a network with 1000 nodes

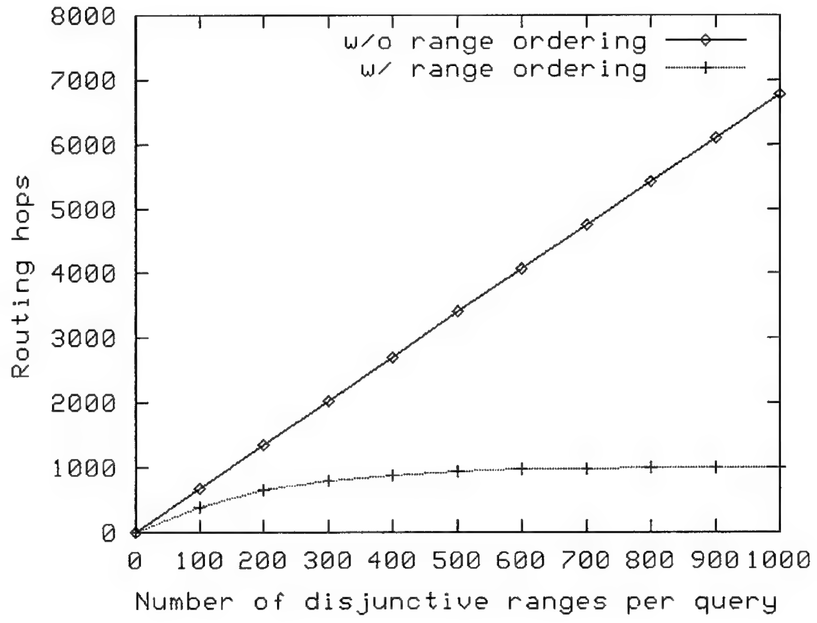


Figure 22: The number of routing hops to resolve disjunctive range queries (0.1% selectivity) in a network with 1000 nodes

Figure 22 shows the result for up to 1000 disjunctive ranges with 0.1% selectivity each in the same network. From these two experiments, we can see that the range ordering algorithm takes less routing hops to resolve a range query than the simple algorithm, and that its routing hops are indeed bounded by *N*.

3.11 Throughput of Triple Storing and Querying

In this section, we present throughput measurements for triple storing operations and query operations in a RDFPeers network deployed on a 16-node cluster. The nodes in the cluster are all dual Pentium III 547 MHz workstations with 1.5 Gigabytes memory, and connected with a 1-Gigabit switch.

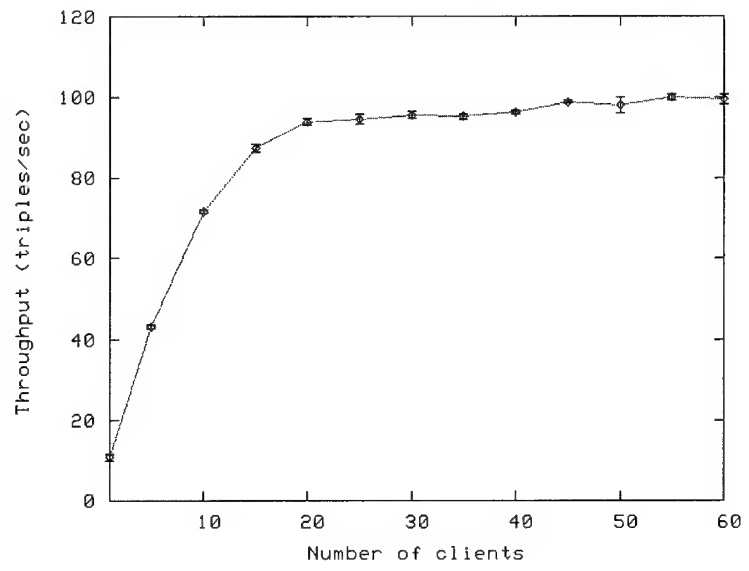


Figure 23: Aggregated throughput of triple storing increases with the number of concurrent clients in a 12-node network

We first measured the aggregated throughput of triple storing in a RDFPeers network with 12 nodes. We increase the number of clients that concurrently store triples into the network from 1 to 25. Figure 23 shows that the number of triples stored per second of all clients increases sub-linearly with respect to the number of clients. When there is only one client, it stores 10.74 triples per second. However, 25 clients can concurrently store 94.00 triples per second. When the number of clients is more than 25, the throughput does not increase significantly because of the computation and network limitation of our fixed number of nodes.

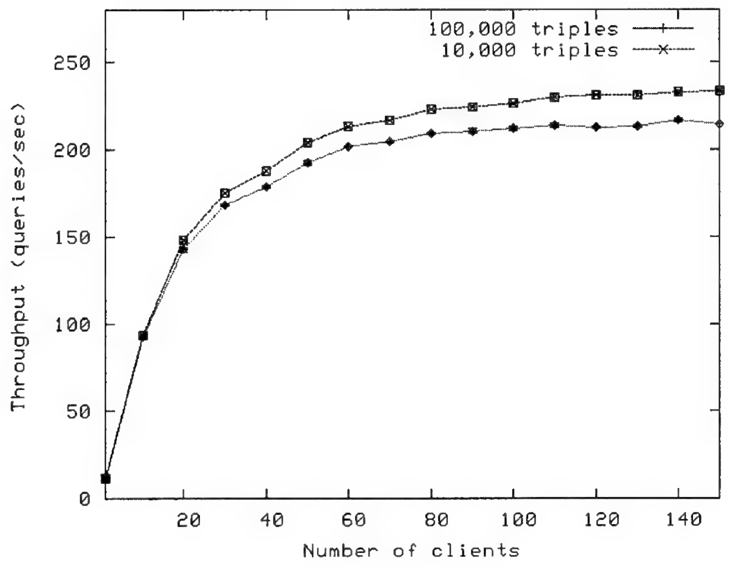


Figure 24: Aggregated query throughput increases with the number of concurrent clients in a 12-node network with 10,000 and 100,000 preloaded respectively

We then measured the aggregated query throughput for the same RDFPeers network that preload 10,000 and 100,000 triples respectively at the beginning of the test. Each client performed 200 queries on one RDFPeers node simultaneously and the total rate of queries is calculated. Figure 24 shows that the query rates of two configurations both increase sub-linearly with respect to the number of clients. When there is only one client, the query rates are 11.08 and 11.47 queries per second respectively for 10,000 and 100,000 preloaded triples. While for 150 clients, the query rates are 214.38 and 233.20 queries per second respectively for 10,000 and 100,000 triples. Similarly to storing operations, the query throughput also stops increasing significantly when there are more than 120 clients. These results also show that the query rate only drops slightly when the preloaded triples increase from 10,000 to 100,000.

These throughput results of storing and query operations are still preliminary and not enough for applications that care about high throughput for storing and query triples. We will further do some performance tuning for our RDFPeers implementation, such as using asynchronous socket, customized message marshaling and unmarshaling, and batched triple insertion.

3.11.1 Message Traffic of Subscription and Notification

We performed two experiments, running RDFPeers on the same cluster, with up to ten nodes per machine. In the first experiment, we set up a network of *N* nodes, then inserted 1,024 subscription requests into the network (1,024/*N* subscriptions per node), followed by inserting 16,348 triples into the network (each node inserts 16,348/*N* triples). Each triple matches 8 subscriptions. Figure 25 shows that the subscription messages needed grow logarithmically with the size of the network, 1024 * log(N), while the number of notification messages needed approaches a constant, 16,348 triples times 8 subscriptions each. It is less than that constant for small networks because some subscriptions can be resolved within a single node. The latter are bounded by that constant assuming that the subscription-handling node can store the network address of the subscriber and open a direct connection to notify it, as our implementation does, otherwise, if Chord successor routing is used, the latter number would grow logarithmically with network size as well. Finally, as expected, the cost of inserting triples grows logarithmically with network size, 16,348 triples * the 3 times each triple is indexed * *log N*.

In the second experiment, we kept the number of nodes in the network constant, but varied the percentage of topics that each node subscribes to. As expected, Figure 26 shows that the number of messages needed grows linearly with the subscription rate, both for the subscription traffic and for the notification traffic.

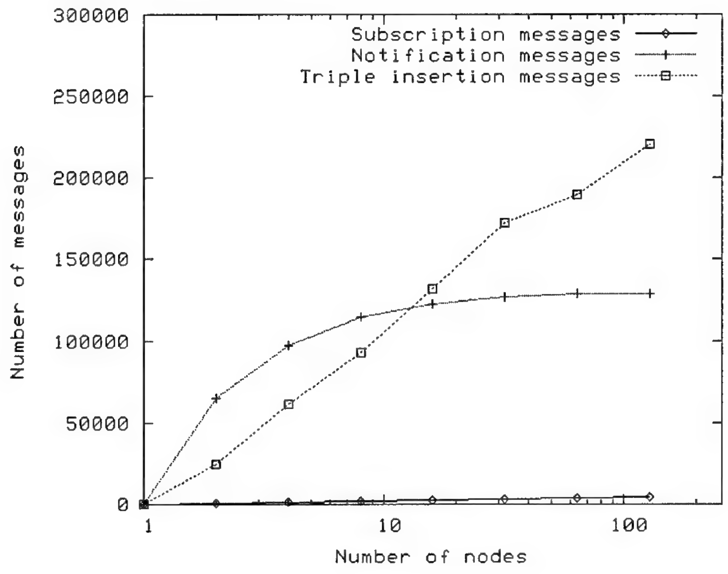


Figure 25: For a constant number of triple subscriptions and insertions, the cost of our subscription scheme in messages grows no more than logarithmically with network size, 128 topics, 1024 subscriptions and 16384 triples

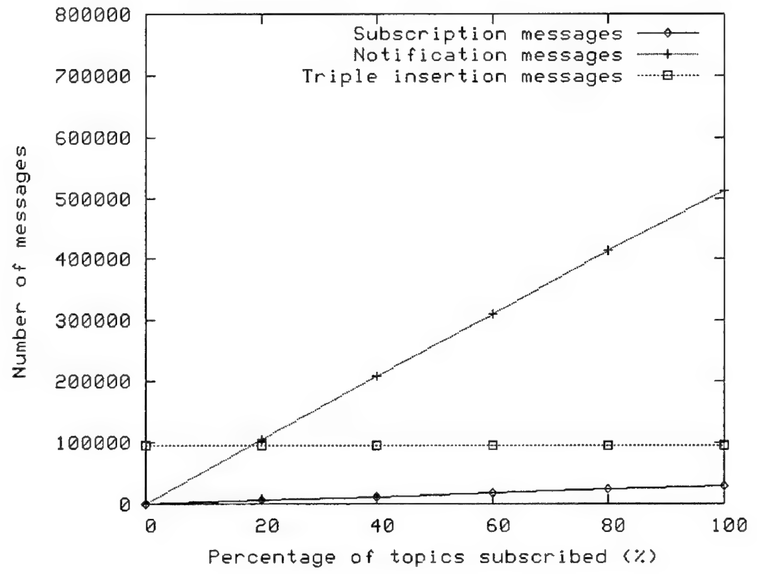


Figure 26: For a constant network size and load, registration and notification traffic grows linearly with the subscription rate, 128 topics, 64 nodes, and 8192 triples

3.12 Dealing with Overly Popular URIs and Literals

Even today's cheapest PCs have a surprising storage capacity, each can store well over ten million RDF triples by dedicating 10 Gigabytes of its typical 80-120 GB disk. Nevertheless, some triples in RDF such as those with the predicate *rdf:type* may occur so frequently that it becomes impossible for any single node in the network to store all of them. That is, in practice, triples may not hash around the Chord identifier circle uniformly due to the non-uniform frequency count distribution of URIs and literals. Figure 27 shows the frequency count distribution of the URIs and Literals in the RDF dump of the ``Kids and Teens" catalog of the Open Directory Project (http://rdf.dmoz.org). There are two RDF files for this catalog: *kt-structure.rdf.u8.gz* and *kt-content.rdf.u8.gz*. The former describes the tree structure of this catalog and contains 19,550 triples. The latter describes all the sites in this catalog and contains 123,222 triples. Figure 27 shows that only 10 to 20 URIs and literals (less than 0.1%) occur more than a thousand times.

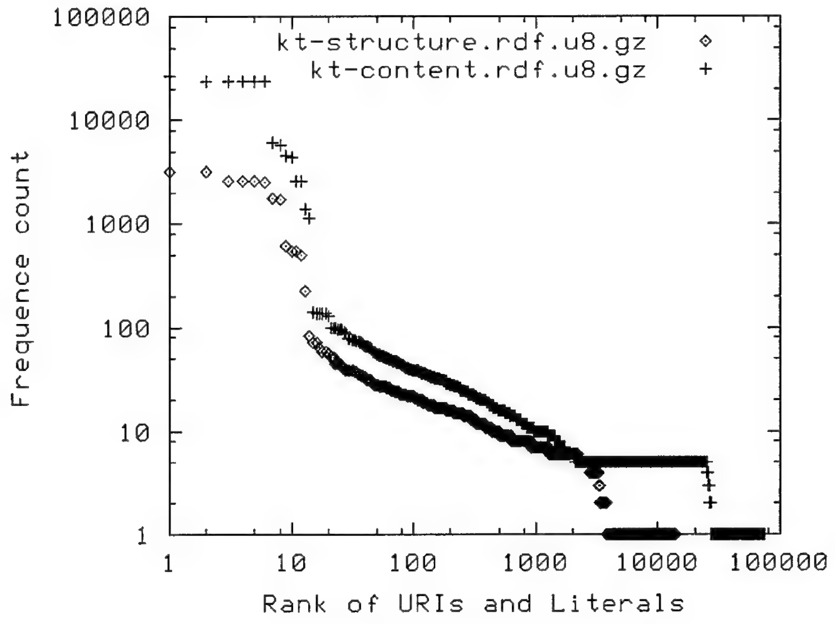


Figure 27: The frequency count distribution of URIs and literals in the ODP Kids and Teens catalog

Figure 28 lists the URIs and literals that occur more than 1000 times in *kt-structure.rdf.u8.gz*. For example, since each URI as a predicate value will be stored at only one node, this node has the global knowledge about the frequency count of this predicate value.

We deal with predicate values that become overly popular by simply no longer indexing triples on them. Each node defines a *Popular_Threshold* parameter based on its local capacity and willingness (subject to some minimum community expectation). Each node keeps counting the frequency of each predicate value. If a predicate value occurs more than *Popular_Threshold* times, the node will refuse to store it and internally makes a note of that. If the node receives a search request with the overly popular value for the predicate, it sends a refusal message back to the requesting node and the requesting node must then find an alternative way of resolving the query by navigating to the target triples though either the subject or object values. This approach will add *O(log N)* to that node's total query cost in hops. We limit subject and object values in the same way. We are aware that this still makes the node with popular URIs a hotspot for query messages that can be addressed by querying nodes caching which queries were refused in the past. In essence, this means that you cannot ask e.g. ``which instances in the world are the subclass of some class". However, these queries are so general and would return so many triples that we suspect they would rarely be of use in practice anyway (in analogy to the English language, where the words ``a" and ``the" occur frequently but provide little value as search terms). For the above query, you could alternatively gather the class URIs for which you want to look for instances for, then traverse to the instances via that set of URIs by issuing a Q4-style query.

Frequency	URI or literal	Type
3158	rdf:type	predicate
3158	dc:Title	object
2612	http://dmoz.org/rdf/Topic	object
2612	http://dmoz.org/rdf/catid	predicate
2574	http://dmoz.org/rdf/lastUpdate	predicate
2540	http://dmoz.org/rdf/narrow	predicate
1782	http://dmoz.org/rdf/altlang	predicate
1717	dc:Description	object

Figure 28: URIs and literals that occur more than one thousand times in *kt-structure.rdf.u8.gz*

Figure 29 shows the minimum, average, and maximum number of triples per node with *Popular_Threshold* from 500 to 32,000. In this experiment, we store both *ktstructure.rdf.u8.gz* and *ktcontent.rdf.u8.gz* (total 142,772 triples) into a network of 100 physical nodes (and the standard Chord log(100)=6 virtual nodes per physical node for trading off load balancing against routing hops). When *Popular_Threshold*=32,000, there are no overly popular URIs or literals being removed and there is an average of 4303 triples per node. However, the load is unevenly balanced - the minimum number of triples per node is 700 while the maximum number of triples per node is 36,871. When *Popular_Threshold* is set to 500, there are 20 overly popular URIs and literals being removed from indexing and there are an average of 2352 triples per node. The minimum number of triples per node is 688 while the maximum number of triples per node is reduced to 4900 - which we believe at less than an order of magnitude difference is acceptable load balancing.

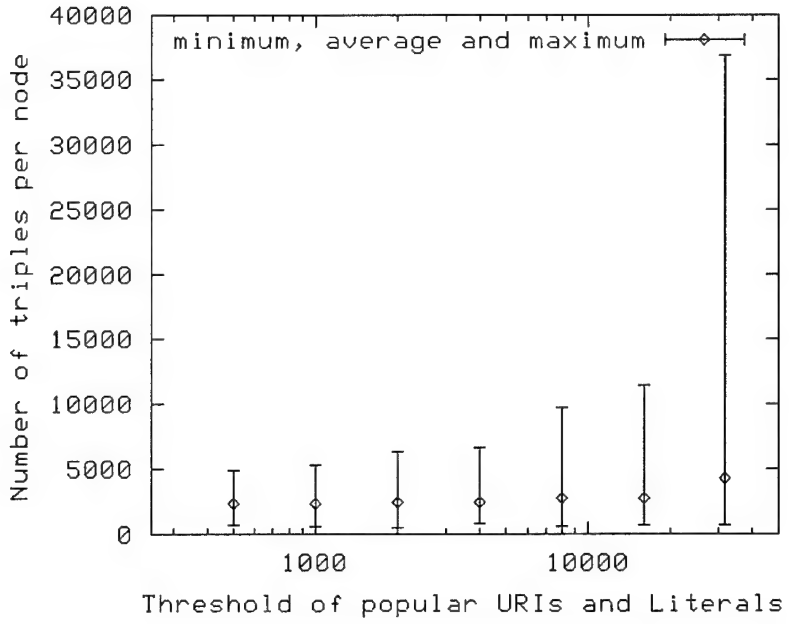


Figure 29: The number of triples per node as a function of the threshold of popular triples (100 physical nodes with 6 virtual nodes per physical node)

3.13 Load Balancing via Successor Probing

Although limiting overly popular URIs and literals greatly reduces the difference between the maximum and minimum number of triples per node, the triples are still not uniformly distributed around all nodes. This is because the frequency count distribution of non-popular URIs and literals remains non-uniform even after removing overly popular values. We propose a preliminary *successor probing* scheme inspired by the ``probe-based" node insertion techniques of [Ghandeharizadeh et al. 2003] to further achieve a more balanced triple storage load on each node. In Chord, the distribution of node identifiers is uniform and independent of the data distribution. In this successor probing scheme, we use a sampling technique to generate a node identifier distribution adaptive to the data distribution. When a node joins the network, it will use SHA1 hashing to generate *Probing_Factor* candidate identifiers. Then it uses Chord's successor routing algorithm to find the successors corresponding to these identifiers. All the successors will return the number of triples that would be migrated to the new node if it joined there, and the new node will choose the identifier that gives it the heaviest load. The cost of this technique is that it increases the insertion time of a triple from *log N* to *Probing_Factor* * *log N*. It is our intuition that *log N* is a good setting for the probing factor.

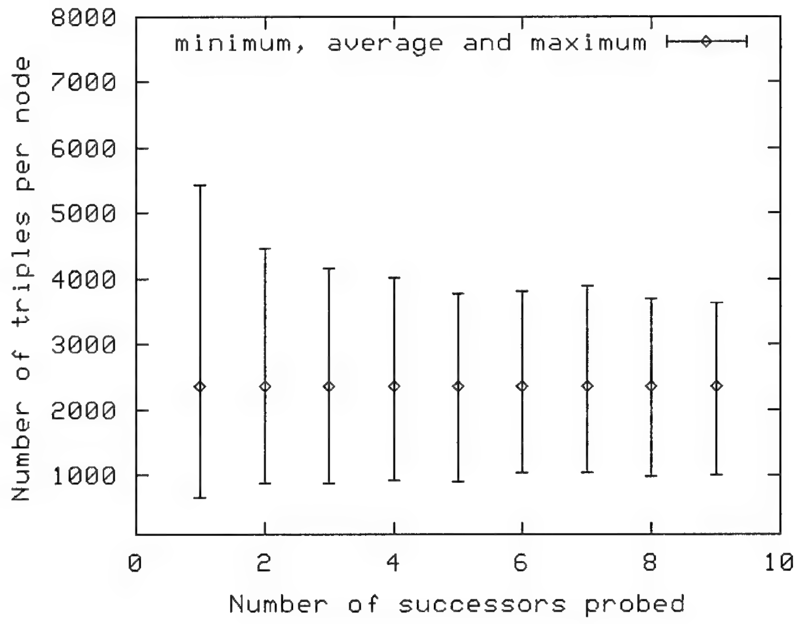


Figure 30: The number of triples per node as a function of the number of successor nodes probed (100 physical nodes, Popular_Threshold=1000)

Figure 30 shows the minimum, average and maximum number of triples per node with *Probing_Factor* from 1 to 9 in a network with 100 physical nodes. The *Popular_Threshold* is set to 1000 in this experiment. If there is no successor probing, the most loaded node has 7.2 times more triples than the least loaded node. If each node probes 9 nodes when it joins, the node with the heaviest load only has 2.6 times more triples than the node with the lightest load - which further reduces load imbalances to much less than an order of magnitude. We can further improve load balancing with a background virtual node migration scheme proposed in [Rao et al. 2003], subject to the limitation that it cannot distribute the load for a single overly popular value.

3.14 Related Work

Our work on RDFPeers was inspired by a number of research traditions including RDF metadata management, structured peer-to-peer systems, and publish/subscribe systems.

3.14.1 RDF Metadata Management Systems

Many centralized RDF repositories have been implemented to support storing, indexing and querying RDF documents, such as RDFDB [Guha], Inkling [Miller], RDFStore (http://rdfstore.sourceforge.net) and Jena [McBride 2001]. These centralized RDF repositories typically use in-memory or database-supported processing, and files or a relational database as the back-end RDF triple store. RDFDB supports a SQL-like query language, while Inkling, RDFStore and Jena all support SquishQL-style RDF query languages. Centralized RDF repositories are very fast and can scale up to many millions of triples. However, they have the same limitations as other centralized approaches, such as a single processing bottleneck and a single point of failure.

To support integrated querying of distributed RDF repositories, Stuckenschmidt et al. (2004) extend the Sesame system to a distributed architecture that introduces a RDF API implementation (Mediator SAIL) on top of the distributed repositories. Their work focuses on the index structure as well as query optimization in the mediator SAIL implementation. This mediator approach can support arbitrary complex queries and works well for small size of data sources. However, it is difficult for this approach to scale up to Internet size of data sources. Edutella [Nejdl 2002] and its successor super-peer based RDF P2P network [Nejdl 2003] were discussed in Section 3.1 . Super-peers are often desirable in order to place the load unevenly among heterogeneous nodes, but our scheme can achieve the same effect more flexibly by nodes hosting more or fewer Chord virtual nodes according to their capacity. REMINDIN [Tempich et al. 2004] developed a lazy learning approach for the SWAP platform [Ehrig at al. 2003] to efficiently route semantic queries based on social metaphors. However, it only learns how to forward simple queries and still lacks efficient algorithm for complex queries.

Much work in the Semantic Web and information integration literature has been emphasized on solving the semantic interoperability problem among data sources with heterogeneous ontologies. ChattyWeb [Aberer et al. 2003] enables the participating data sources to incrementally develop global agreement in an evolutionary and completely decentralized bottom-up process by learning the graph of local mappings among schemas through gossiping. Piazza [Halevy et al. 2003] also eliminates the need for a global mediated schema by describing the mappings between sets of XML and RDF source nodes and evaluating those schema mappings transitively to answer queries. These two systems forward queries to the peers based on schema similarities, which is complementary to RDFPeers that indexes instances of RDF statements. It might be interesting to develop some hybrid systems that leverage schema mapping on the top of RDFPeers.

3.14.2 Structured Peer-to-Peer Systems

Recent structured P2P systems use message routing instead of flooding by leveraging a structured overlay network among peers. These systems typically support distributed hash table (DHT) functionality and offer the operation *lookup (key)*, which returns the identity of the node storing the object with the key [Ratnasamy 2001]. Current proposed DHT systems include Tapestry [Zhao 2001], Pastry [Rowston and Druschel 2001], Chord [Stoica 2001], CAN [Ratnasamy 2001], and Koorde [Kaashoek and Karger 2003].

These DHT systems provide scalable distributed lookup for unique keys. However they can not support efficient search, such as keyword search and multi-dimensional range queries. Reynolds and Vahdat (2003)] proposed an efficient distributed keyword search system, which distributes an inverted index into a distributed hash table, such as Chord or Pastry. To minimize the bandwidth consumed by multi-keyword conjunctive searches, they use bloom filters to compress the document ID sets by about one order of magnitude and use caching to exploit temporal locality in the query workload. For large sets of search results, they also use streaming transfers and return only the desired number of results. pSearch [Xu and Dwarkadas 2003] is another peer-to-peer keyword search system that distributes document indices into a CAN network based on the document semantics generated by Latent Semantic Indexing (LSI). It uses content-aware node bootstrapping to force the distribution of nodes in the CAN to follow the distribution of indices.

Andrzejak and Xu et al. (2002) extend CAN for handling range queries on single attributes by mapping one dimensional space to CAN's multi-dimensional space using Hibert Space Filling Curve as hash function. However, this work did not address multi-attribute range queries. In contrast to Andrzejak's system, Schmidt and Parashar (2003) proposed a dimension reducing indexing scheme that efficiently maps the multi-dimensional information space into the one dimensional Chord identifier space by using Hibert Space Filling Curve. This system can support complex queries containing partial keywords, wildcards, and range queries. PIER [Heubsch et al. 2003] focuses on design a massively distributed query engine based on DHT systems, especially for distributed equi-joins. Their join algorithms are based on a multicast primitive that flood the query to all nodes in the same namespace. However, PIER does not support efficient range predicates because DHTs are a hashing mechansim. Actually the work in PIER is complementary to RDFPeers for supporting efficient join operations.

3.14.3 Publish/Subscribe Systems

Besides RDFPeers, there are several other distributed RDF metadata management system that provides publish and subscribe mechanisms. MDV [Keidl 2002] is a distributed RDF metadata management system based on a 3-tier architecture and supports caching and replication in the middle-tier. It implemented a filter algorithm based on relational database technology that efficiently computes all subscribers for created, updated and deleted RDF data. Chirita et al (2004) proposed a peer-to-peer RDF publish/subscribe system that was based on a super-peer based RDF peer-to-peer network. In contrast to RDFPeers, subscriptions in this approach are selectively broadcast to other super-peers based on their advertisements, while subscriptions in RDFPeers are routed to and store on a particular node that is also responsible for storing matching RDF statements.

Publish/subscribe systems have also been studied extensively in the networking and distributed systems literature [Carzaniga et al. 2001, Castro et al. 2002, the CORBA Notification Service 1.0.1, the Java Distributed Event Specification, the Web Services Notification standard]. However, those systems typically only support topic-based or type-based subscriptions. In contrast, RDFPeers and other metadata publish/subscribe systems allow more expressive subscriptions for metadata. Recent advances in distributed hash tables also enable a new class of scalable publish/subscribe systems [Castro et al. 2002, Tam et al. 2003] that do not rely on a centralized server nor on subscription broadcasting.

3.15 Conclusion, Robust P2P Knowledge Storage

We would like to implement the RDQL-to-RDFPeers native queries translator that we have only sketched in this paper, and improve load balancing using a background virtual node migration scheme. We would also like to take more measurements of the scalability of our subscription design, such as throughput stress tests: ``For what number of subscribers (UCAVs), subscribing to how much content, does the scheme break, not just in terms of abstract routing hops, but in reality for our actual Java implementation?".

In conclusion, RDFPeers provides efficient distributed RDF metadata storage, query and subscription in a structured P2P network. It avoids flooding queries to the network and guarantees that query results will be found if they exist. RDFPeers can also balance the triple-storing load between the most and least loaded nodes by using successor probing scheme. Its state cost in neighborhood connections is logarithmic to the number of nodes in the network, and so is its processing cost in routing hops for all insertion, most query and subscription operations. RDFPeers offers subscriptions that, assuming a fixed number of subscriptions per node, scale to networks of many nodes. RDFPeers also preserves subscriptions as well as the original data by replicating content to a fixed number of nearby nodes so that the network can repair itself without data loss when a node (UCAV) suddenly dies. RDFPeers thus enables fault-tolerant distributed RDF repositories of truly large numbers of participants. We hope it can become the basis for a new type of metadata-driven and egalitarian community applications on the Internet.

4 Transitions to Military Applications

A MARBLES sister project delivers the SNAP flight scheduling application, which includes software produced by the MARBLES project. SNAP is fielded at the following locations:

Marine Air Group 13 in Yuma. The system was fielded in all 4 squadrons.

Marine Expeditionary Units. The system was fielded on board the USS Bonhomme Richard, the USS Belleau Wood, the USS Pelleliu and the USS Essex that conducted operations in Iraq, Japan and Afghanistan.

The software produced by the MARBLES sister project involves over 500,000 lines of code. Of these about 40% is part of the generic negotiation framework, and 60% is application specific; the MARBLES project contributed about 15,000 lines of the code to the former. The rest of this section provides a short overview of the SNAP application that embodies MARBLES-contributed code.

Marine Corps Harrier squadrons consist of two planning units – operations (what to fly) and maintenance (how to schedule work on the aircraft to support the operations). SNAP addresses the operational side of flight scheduling (a sister application from Vanderbilt University addresses the maintenance side, which is not further discussed here).

The commanding officer of the squadron defines the overall schedule goals for a given planning horizon. For example, for the squadron to participate in a certain training exercise, a possible short-term goal is to have Smith and Jones obtain their night systems qualification, to fly 20 sorties per day and to maintain flight equity (meaning that every pilot obtains roughly the same number of flight hours). The commanding officer also gives yearly and monthly guidance, for example how much fuel and ammunition should be expended.

The operations officer and his/her staff then produce weekly schedules to meet the guidance, and refine those weekly schedules every day to produce the daily schedules that actually get executed. The operations and maintenance office communicate frequently to coordinate their schedules: in order for the operations office to produce a schedule they need to know how many aircraft of each type are available. In order to answer that question the maintenance office needs to know the flight schedule in order to figure out whether they have time to carry out all the usage-based and calendar-based maintenance that must be done on the aircraft. (Usage-based maintenance is performed after the aircraft has accumulated a number of flight hours; calendar-based maintenance is performed after a predefined number of days pass – even if the aircraft has not been flown at all.) The cycle is broken by starting with estimates and refining those estimates though iterative refinement of the schedules.

4.1.1 Manual Harrier Flight Scheduling Today

The users have databases or paper manuals that provide all the information relevant for producing the schedules. The Operations office of a squadron has databases that record the flight logs of each pilot, their qualifications, etc. The Maintenance office has databases that record all the maintenance work items that must be performed or are being performed on each aircraft.

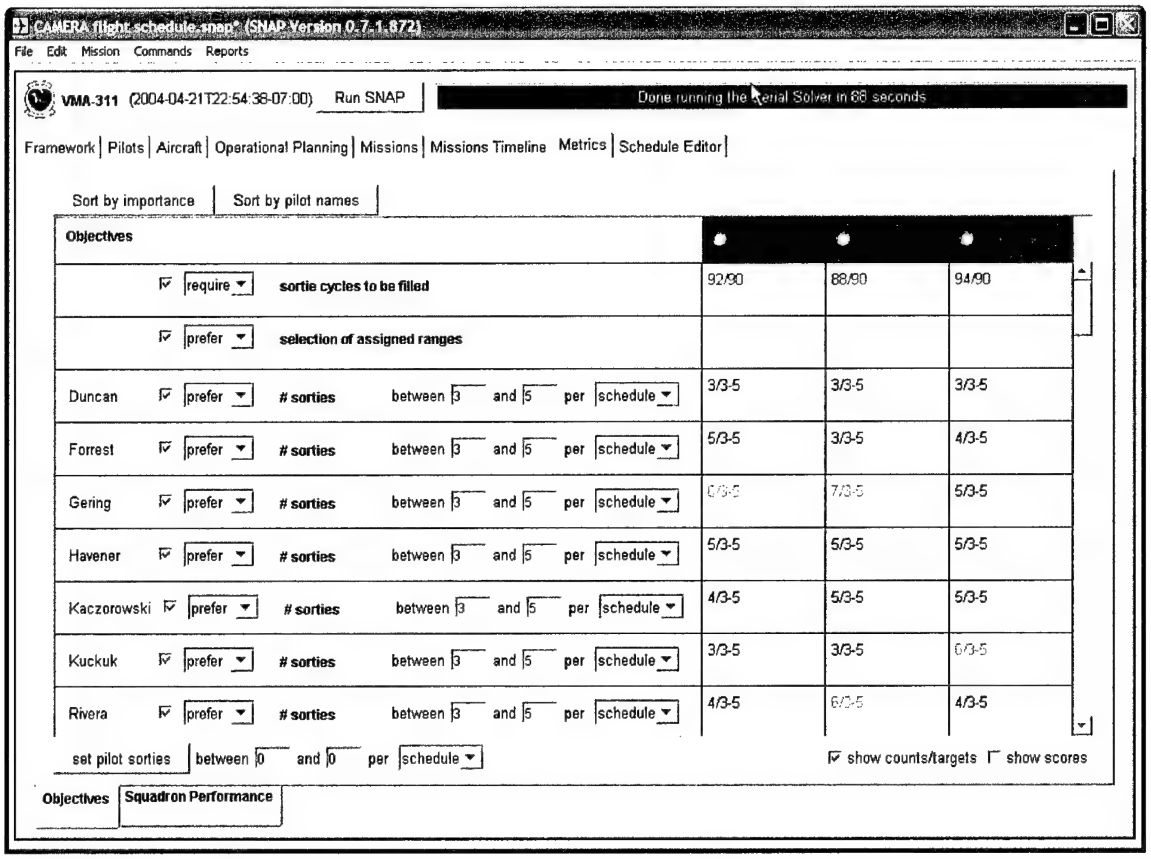
These databases have viewers and editors that allow users to see what is happening, and to edit the information. The Operations office has a sophisticated schedule editor application that enables users to enter and format flight schedules, but they do not have an application that *computes* a schedule. The users determine the flight schedules manually, typically on a white board, and then enter it in the computer to print the official schedules that get signed by the commanding officers. The maintenance schedules are kept on a white board and on paper and are currently never entered in a computer.

4.1.2 The Challenge in Automating the Flight Schedule

The challenge is to automate the production of the operations and maintenance schedules. The payoff is large because developing the schedules by hand is labor intensive and time consuming (typically 6 hours for a weekly operations schedule), and little time is left to explore alternatives and to deal with often changing requirements. In addition, producing schedules that extend beyond a week is infeasible, resulting in commanders having limited ability to forecast the consequences of taking on new commitments (e.g., can you participate in a week-long exercise at the beginning of next month and still make your deployment commitments 6 months from now?)

4.1.3 Highlights of the Automated Flight Scheduling System

Figure 31 shows a screen shot of the display where operators can specify the last three types of goals (in addition to variations of these goals). Each row corresponds to a separate goal (also referred to as an objective). The table at the right of the image shows a history of how well different versions of the schedules satisfy the goals. In this example the operator is getting close to a satisfactory schedule. The schedule contains 2 more sorties than desired, and one pilot is flying too much.


CAMERA flight schedule.snap* (SNAP Version 0.7.1.872)
File Edit Mission Commands Reports
VMA-311 (2004-04-21T22:54:38-07:00) Run SNAP
Done running the Serial Solver in 88 seconds
Framework | Pilots | Aircraft | Operational Planning | Missions | Missions Timeline Metrics | Schedule Editor
Sort by importance | Sort by pilot names
Objectives | Squadron Performance

Objectives						
	☑ require	sortie cycles to be filled		92/90	88/90	94/90
	☑ prefer	selection of assigned ranges				
Duncan	☑ prefer	# sorties	between 3 and 5 per schedule	3/3-5	3/3-5	3/3-5
Forrest	☑ prefer	# sorties	between 3 and 5 per schedule	5/3-5	3/3-5	4/3-5
Gering	☑ prefer	# sorties	between 3 and 5 per schedule	6/3-5	7/3-5	5/3-5
Havener	☑ prefer	# sorties	between 3 and 5 per schedule	5/3-5	5/3-5	5/3-5
Kaczorowski	☑ prefer	# sorties	between 3 and 5 per schedule	4/3-5	5/3-5	5/3-5
Kuckuk	☑ prefer	# sorties	between 3 and 5 per schedule	3/3-5	3/3-5	6/3-5
Rivera	☑ prefer	# sorties	between 3 and 5 per schedule	4/3-5	6/3-5	4/3-5

set pilot sorties | between 0 and 0 per schedule

☑ show counts/targets ☐ show scores

Figure 31: Fine-tuning scheduling preferences in the Metrics screen

The objective function is a linear combination of metrics that specify how well each goal is achieved. Each goal is scored using a piece-wise linear function that specifies how good it is to obtain a certain quantity of something. For example, an objective function for the number of sorties goal could be specified as follows: flying 0 sorties gives a score of 0, flying 90% of the sorties gives a score of 0.5, and flying 100% gives a score of 1.

The score of a schedule is specified as a linear combination of the score for each goal. Note: in the implemented system, the user interface did not allow users to give numeric weights. Instead we offered the values "require", "prefer" and "don't care", which were defined in such a way that all the "prefers" weighed less than a single "require" (see Figure 31).

One of the lessons learned in the logistics challenge problem is that it is very difficult for users to specify an objective function that captures all the issues they care about. We found that it was only after seeing a solution that users would think about trade-offs that would be impractical to specify in advance (e.g., I'll accept having Jones fly 8 rather than 5 sorties if that is the only way I can get Smith's night systems qualification done).

The structure of a task is shown in Figure 32, which shows the segments and the resources that participate in a task. The segments are shown as a time line at the top of the image. Each segment has a name and duration. For example, the first segment is the "brief" segment during which the pilots attend a briefing on the mission they are about to perform. Its duration is 120 minutes. Each row in the image represents a resource that participates in the task. The first two rows labeled "Lead" and "Wing" represent the pilots. The green bars represent the segments during which resources are needed. For example, the pilots are needed during all segments, whereas the aircraft are needed only after the briefing segment.

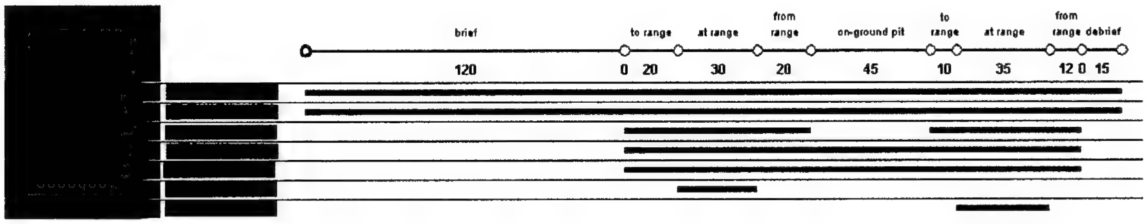
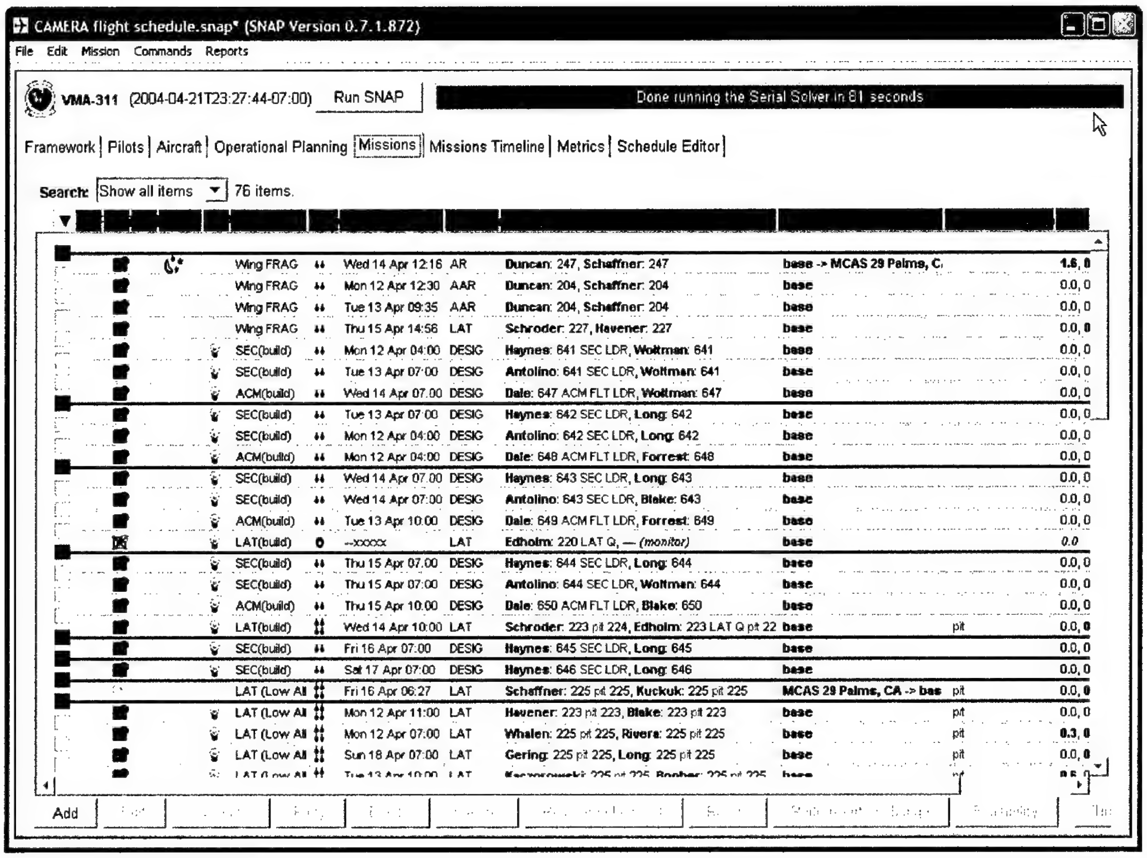


Figure 32: The interactive time-line display for fine-tuning mission segments

Figure 33 shows the results of running the scheduler. Scheduled tasks are marked with a green icon, whereas tasks that the system was unable to schedule are highlighted with a red "X". Also, the resources and times that the system assigned to the tasks are shown in green. Users are free to override any system decision by selecting a task and invoking the task editor.


CAMERA flight schedule.snap* (SNAP Version 0.7.1.872)

File Edit Mission Commands Reports

VMA-311 (2004-04-21T23:27:44-07:00) Run SNAP — Done running the Serial Solver in 81 seconds

Framework | Pilots | Aircraft | Operational Planning | Missions | Missions Timeline | Metrics | Schedule Editor

Search: Show all items — 76 items.

Wing FRAG	Wed 14 Apr 12:16	AR	**Duncan**: 247, **Schaffner**: 247	**base -> MCAS 29 Palms, C**		**1.6, 0**
Wing FRAG	Mon 12 Apr 12:30	AAR	**Duncan**: 204, **Schaffner**: 204	**base**		0.0, 0
Wing FRAG	Tue 13 Apr 09:35	AAR	**Duncan**: 204, **Schaffner**: 204	**base**		0.0, 0
Wing FRAG	Thu 15 Apr 14:56	LAT	**Schroder**: 227, **Havener**: 227	**base**		0.0, 0
SEC(build)	Mon 12 Apr 04:00	DESIG	**Haynes**: 641 SEC LDR, **Woltman**: 641	**base**		0.0, 0
SEC(build)	Tue 13 Apr 07:00	DESIG	**Antolino**: 641 SEC LDR, **Woltman**: 641	**base**		0.0, 0
ACM(build)	Wed 14 Apr 07.00	DESIG	**Dale**: 647 ACM FLT LDR, **Woltman**: 647	**base**		0.0, 0
SEC(build)	Tue 13 Apr 07:00	DESIG	**Haynes**: 642 SEC LDR, **Long**: 642	**base**		0.0, 0
SEC(build)	Mon 12 Apr 04:00	DESIG	**Antolino**: 642 SEC LDR, **Long**: 642	**base**		0.0, 0
ACM(build)	Mon 12 Apr 04:00	DESIG	**Dale**: 648 ACM FLT LDR, **Forrest**: 648	**base**		0.0, 0
SEC(build)	Wed 14 Apr 07.00	DESIG	**Haynes**: 643 SEC LDR, **Long**: 643	**base**		0.0, 0
SEC(build)	Wed 14 Apr 07:00	DESIG	**Antolino**: 643 SEC LDR, **Blake**: 643	**base**		0.0, 0
ACM(build)	Tue 13 Apr 10:00	DESIG	**Dale**: 649 ACM FLT LDR, **Forrest**: 649	**base**		0.0, 0
LAT(build)	--xxxxx	LAT	**Edholm**: 220 LAT Q, — *(monitor)*	**base**		*0.0*
SEC(build)	Thu 15 Apr 07.00	DESIG	**Haynes**: 644 SEC LDR, **Long**: 644	**base**		0.0, 0
SEC(build)	Thu 15 Apr 07:00	DESIG	**Antolino**: 644 SEC LDR, **Woltman**: 644	**base**		0.0, 0
ACM(build)	Thu 15 Apr 10:00	DESIG	**Dale**: 650 ACM FLT LDR, **Blake**: 650	**base**		0.0, 0
LAT(build)	Wed 14 Apr 10:00	LAT	**Schroder**: 223 pit 224, **Edholm**: 223 LAT Q pit 22	**base**	pit	0.0, 0
SEC(build)	Fri 16 Apr 07:00	DESIG	**Haynes**: 645 SEC LDR, **Long**: 645	**base**		0.0, 0
SEC(build)	Sat 17 Apr 07:00	DESIG	**Haynes**: 646 SEC LDR, **Long**: 646	**base**		0.0, 0
LAT (Low Al	Fri 16 Apr 06:27	LAT	**Schaffner**: 225 pit 225, **Kuckuk**: 225 pit 225	**MCAS 29 Palms, CA -> bas**	pit	0.0, 0
LAT (Low Al	Mon 12 Apr 11:00	LAT	**Havener**: 223 pit 223, **Blake**: 223 pit 223	**base**	pit	0.0, 0
LAT (Low Al	Mon 12 Apr 07:00	LAT	**Whalen**: 225 pit 225, **Rivera**: 225 pit 225	**base**	pit	**0.3, 0**
LAT (Low Al	Sun 18 Apr 07:00	LAT	**Gering**: 225 pit 225, **Long**: 225 pit 225	**base**	pit	0.0, 0

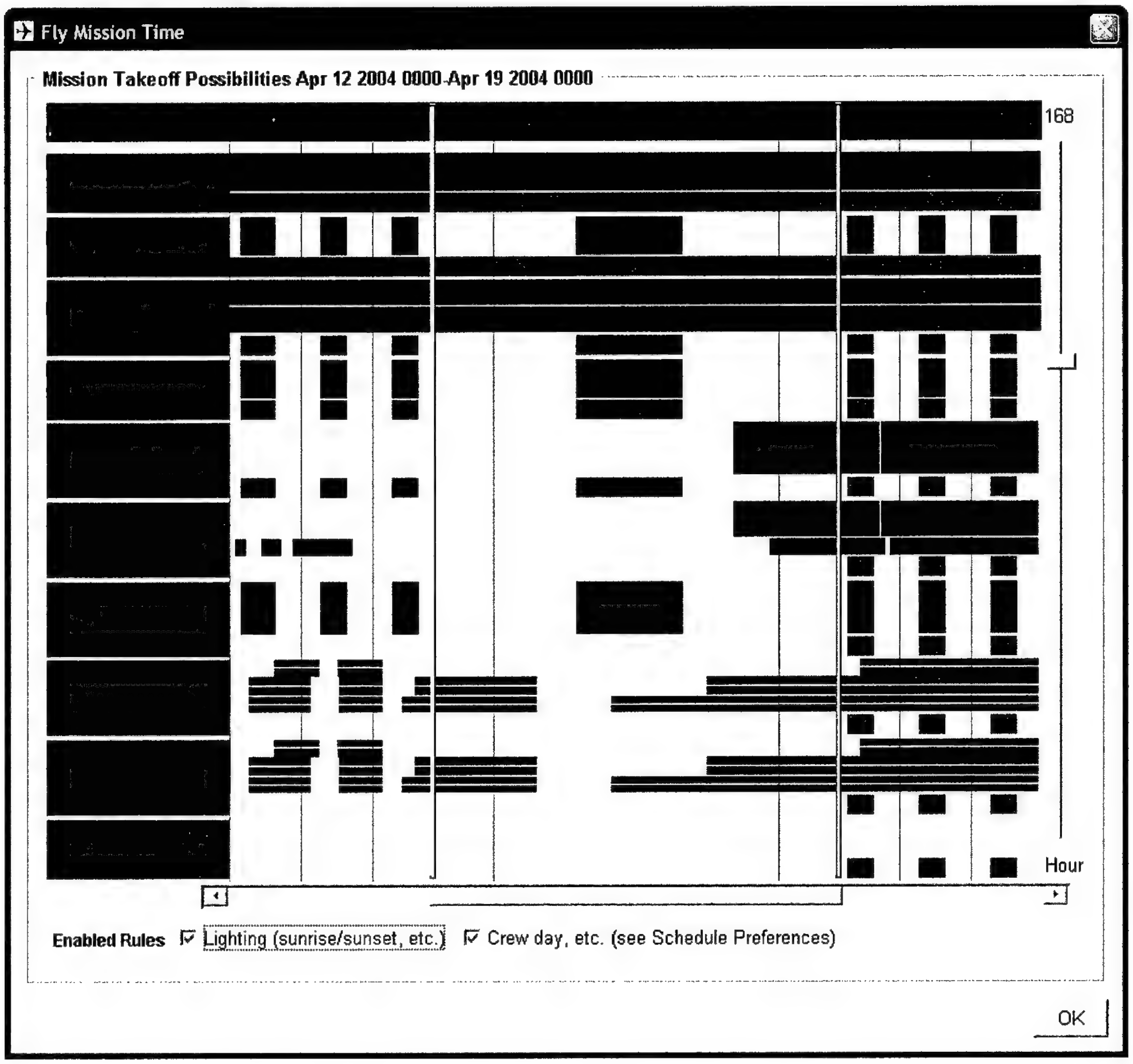
Figure 33: The resulting overall schedule (system choices are in green)

When a task fails to schedule, the user has to go back to step 1 (Specify the problem) to modify the problem so that the solution that the system can produce is satisfactory for the user. We call this iterative step problem reformulation.

There are typically many ways to reformulate a problem to achieve a satisfactory solution. The choice depends on the trade-offs that users are willing to make. Users can make more resources available, they can override certain constraints that prevent a task from scheduling (e.g., in the previous image the user waived the light constraints on the first task, as indicated by the moon and stars icon), they can change priorities, they can scale back their goals, etc. The system provides extensive editing capabilities that allow users substantial freedom in reformulating their problem.

When a task fails to schedule, the user can invoke the task feasibility display to understand why the task failed to schedule and what could be done to allow it to schedule. Figure 34 shows the feasibility display. It contains a row for each resource requirement of a task and for the constraints that it must satisfy. The planning horizon for the schedule is shown horizontally. Each point on the horizontal axis represents a potential start time (e.g., take-off time of a flight) for a task. The green bars in a row represent possible task start times when resources listed in that row and the rows above are available. A red bar appears in a row when that row is the first row that causes some possible start times to be eliminated. The blue bars represent the start times when a particular resource is available. For example, the first row has uninterrupted blue and green bars. This means that the Mission Specified Time does not restrict the start time of the task at all. In other words, the user left the start time open. The second row corresponding to Fly Day Times has smaller blue bars. This is because the Fly Day specifies the times of day when flight operations are to be conducted and thus only start times that allow a task to start and end within those hours are allowed. The red bars in the Fly Day Times indicate possible start times that got eliminated because of the Fly Day restrictions. Similarly, going down the display it is easy to see that no pilot was available to fill the Lead position of the task until sometime late on Thursday. The Lead eliminated all possible start times on Monday through Thursday, and allows start times only on Friday, Saturday or Sunday. Looking further down in the display one can see that the task did not get scheduled because no appropriate ranges could be found (see last row).

The feasibility display is very powerful. For example, one can see that in order to schedule the task earlier during the week several things need to happen (in addition to making an appropriate range available). Wednesday is bad because no appropriate pilots can be found to fill the Lead or Wing positions. Tuesday is a good possibility because only the Lead is missing. Monday would work too, but might be harder to pull off because Wing pilots are only available for part of the day.


Fly Mission Time

Mission Takeoff Possibilities Apr 12 2004 0000-Apr 19 2004 0000

Enabled Rules ☑ Lighting (sunrise/sunset, etc.) ☑ Crew day, etc. (see Schedule Preferences)

Figure 34: Understanding scheduling possibilities via the feasibility display

Producing a satisfactory schedule often involves several reformulations in practice. 10 reformulations for a weekly schedule are typical. Given that the scheduler can compute a weekly schedule in about 1 minute, the process can be completed in 15 to 20 minutes (it takes about 6 hours to complete a schedule without the system, so there is no way to achieve a schedule of the same quality without SNAP).

In summary of this section on SNAP, we are proud that the MARBLES project not only resulted in novel published research (see Section 1 for the publication record), but also contributed software to a fielded military application.

5 Acknowledgements

The successor probing technique of Section 3.13 was inspired by discussions with Shahram Ghandeharizadeh and Antonios Daskos about load balancing techniques. We are grateful to Ann Chervenak and Mats Rynge for their support of our work on the Center For Grid Technologies computing cluster. We thank Sameer Maggon for implementing the GUI of Shared-HiKE. We also gratefully acknowledge feedback from Stefan Decker and Geoff Pike. Last but not least, we thank the Air Force Office of Scientific Research for supporting our work.

6 References

Aberer, K., Cudré-Mauroux, P., and Hauswirth, M. The chatty web: Emergent semantics through gossiping. In *the 13th World Wide Web Conference (WWW2003)*, May 2003.

Andersson, M., and Sandholm, T. 1999. Time-Quality tradeoffs in reallocative negotiation with combinatorial contract types. In *Proceedings of AAAI-99*, Orlando, Florida.

Andrzejak, A. and Xu, Z.. Scalable, efficient range queries for grid information services. In *Second IEEE Int'l Conference on Peer-to-Peer Computing (P2P2002)*, Sep. 2002.

Atkins, E.; T. Abdelzaher; Shin, K.; and E. Durfee, E. 1999. Planning and resource allocation for hard real-time. In *Proceedings of the Third International Conference on Autonomous Agents,* Seattle, Washington.

Boutilier, C.; Goldzmidt, M.; and Sabata, B. 2000. Sequential auctions for the allocation of resources with complementarities. In *Proceedings of IJCAI-99*, Stockholm, Sweden.

Cai, M., Shahram Ghandeharizadeh, S., Schmidt, R., and Song, S. A Comparison of Alternative Encoding Mechanisms for Web Services. Proceedings of the 13th International Conference on Database and Expert Systems Applications (DEXA), 10 pages, (Aix en Provence, France, September 2-6) 2002.

Cai, M., **Frank**, M., Chen, J., and P. **Szekely**. MAAN: A multi-attribute addressable network for grid information services. In *4th Int'l Workshop on Grid Computing*, 2003a.

Cai, M., **Frank,** M., Chen, J., **Szekely,** P. MAAN: A Multi-Attribute Addressable Network for Grid Information Services. 4th International Workshop on Grid Computing (Grid2003), 8 pages, (Phoenix, Arizona, Nov 17) 2003b.

Cai, M. and **Frank,** M.. RDFPeers: A Scalable Distributed RDF Repository based on A Structured Peer-to-Peer Network University of Southern California - Computer Science Technical Report 03-807, December 2003c. (cite the WWW2004 paper below instead, this is the version of the paper as it was submitted)

Cai, M., Chervenak, A., and **Frank**, M. A peer-to-peer replica location service based on a distributed hash table. In *the 2004 ACM/IEEE Conference on Supercomputing*

(SC2004), 2004a.

Cai, M. and **Frank,** M. RDFPeers: A Scalable Distributed RDF Repository based on A Structured Peer-to-Peer Network. 13th International World Wide Web Conference (WWW2004), 8 pages, (New York, May 17-22) 2004b. [14.6% acceptance rate, 74 out of 506]

Cai, M., **Frank,** M., Chen, J., **Szekely**, P. MAAN: A Multi-Attribute Addressable Network for Grid Information Services. Journal of Grid Computing, Kluwer, 2004c (accepted for publication on April 6, 2004).

Cai, M. and **Frank,** M.. A Scalable and Subscribable Peer-to-Peer RDF Repository for Distributed Metadata Management. Journal of Web Semantics, 2004d (status: accepted with minor revisions)

Carzaniga, A., Rosenblum, D. S., and Wolf, A. L. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332-383, 2001.

Castro, M., Druschel, P., Kermarrec, A.-M., and Rowstron, A. Scribe: A large-scale and decentralised application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communications (JSAC)*, 2002.

Chen, J., Bugacov, A., **Szekely**, P., **Frank**, M., **Cai**, M., Kim, D. and Robert Neches, R. Distributed Resource Allocation: Knowing When To Quit. *Representations and Approaches for Time-Critical Decentralized Resource/Role/Task Allocation Workshop of the Second International Joint Conference on Autonomous Agents & Multi-Agent Systems* (AAMAS 2003). Melbourne, Australia (July 14) 2003.

Chirita, P. A., Idreos, S., Koubarakis, M., and Nejdl, W. Publish/subscribe for RDF-based P2P networks. In *the 1st European Semantic Web Symposium (ESWS 2004)*, May 2004.

Choy, M., and Singh, A. 1992. Efficient fault tolerant algorithms in distributed systems. In *24th ACM Symposium on Theory of Computing*, pp. 593–602.

Collins, J.; Sundareswara, R.; Tsvetovat, M.; Gini, M.; and Mobasher, B. 1999. Search strategies for bid selection in multiagent contracting. In *JCAI-99 Workshop on Agent-mediated Electronic Commerce* (AmEC-99).

Clark, D. D. The design philosophy of the DARPA internet protocols. In ACM *SIGCOMM*, pages 106-114, Stanford, CA, Aug. 1988.

Decker, S. and **Frank,** M. The Networked Semantic Desktop, WWW'2004 Workshop on Application Design, Development and Implementation Issues in the Semantic Web, May 18th, 2004.

Deelman, E., Singh, G., Atkinson, M. P., Chervenak, A. , Hong, N. P. C., Kesselman, C. , Patil, S. , Pearlman, L. , and shi Su, M. Grid-based metadata services. In *16th International Conference on Scientific and Statistical Database Management (SSDBM04)*, June 2002.

Ehrig, M. , Haase, P. , Staab, S. , and Tempich, C. Swap: A semantics-based peer-to-peer system. In *JXTA Workshop*, November 2003.

Ferguson, D.; Nikolaou, C.; Sairamesh, J.; and Yemini, Y. 1996. Economic models for allocating resources in computer systems. In S. Clearwater (Ed.), *Market-Based Control: A Paradigm for Distributed Resource Allocation*. Hong Kong: World Scientific.

Frank, M.; Bugacov, A.; Chen, J.; Dakin, G.; **Szekely**, P.; and Neches, R. "The Marbles Manifesto: A Definition and Comparison of Cooperative Negotiation Schemes for Distributed Resource Allocation," Proceedings of the *2001 AAAI Fall Symposium on Negotiation Methods for Autonomous Cooperative Systems*, November 2-4, 2001, North Falmouth, Massachusetts.

Frank, M. About New Global-Scale Collaborative Applications enabled by Structured P2P Networks for RDF. (peer-reviewed workshop position paper) World Wide Web Conference Workshop on Semantics in Peer-to-Peer and Grid Computing, 2 pages, 2004.

Garey, M., and Johnson, D. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. San Francisco, CA: W.H. Freeman and Company. Gent, I., and Walsh, T. 1993. Towards an Understanding of Hill-Climbing Procedures for SAT. In *Proceedings of AAAI-93*, pp. 28-33.

Ghandeharizadeh, S. , Daskos, A. , and An, X. PePeR: A distributed range addressing space for P2P systems. In *Int'l Workshop on Databases, Information Systems, and Peer-to-Peer Computing (at VLDB)*, 2003.

O. M. Group, 2002. CORBA Notification Service 1.0.1.

Ghandeharizadeh, S., Papadopoulos, C., **Cai, M.**, and Krishna K. Chintalapudi, K.. Performance of Networked XML-Driven Cooperative Applications. Proceedings of the 2nd International Workshop on Cooperative Internet Computing, 9 pages, (Hong Kong, China, August 18-19) 2002.

Shahram Ghandeharizadeh, Craig A. Knoblock, Christos Papadopoulos, Cyrus Shahabi, Esam Alwagait, Jose Luis Ambite, Min **Cai**, Ching-Chien Chen, Parikshit Pol, Rolfe Schmidt, Saihong Song, Snehal Thakkar, Runfang Zhou. Proteus: A System for Dynamically Composing and Intelligently Executing Web Services. First International Conference on Web Services (ICWS), 5 pages, (Las Vegas, Nevada, June 23-26) 2003.

Halevy, A., Ives, Z., Tatarinov, I. , and Mork, P. Piazza: Data management infrastructure for semantic-web applications. In *the 13th World Wide Web Conference (WWW2003),* May 2003.

Huebsch, R. , Hellerstein, J. M., Lanham, N. , Loo, B. T., Shenker, S. , and Stoica, I. Querying the internet with pier. In *the 29th International Conference on Very Large Data Bases (VLDB2003)*, September 2003.

IBM, Akamai, HP, and S. et al, 2004. Web Services Notification.

S. M. Inc., 1998. Java Distributed Event Specification.

Jackson. We used Dr. D. Jackson's Java implementation of WSAT available at http://sdg.lcs.mit.edu/walksat

Kaashoek, F. and Karger, D. R. Koorde: A simple degree-optimal hash table. In *2nd Int'l Workshop on P2P Systems*, Feb. 2003.

Karvounarakis, G. , Alexaki, S. , Christophides, V. , Plexousakis, D. , and Scholl, M. RQL: A declarative query language for RDF. In *11th World Wide Web Conference*, 2002.

M. Keidl, A. Kreutz, and A. Kemper. A publish and subscribe architecture for distributed metadata management. In *18th International Conference on Data Engineering (ICDE'02)*, February 2002.

Kirkpatrick, S.; Gelatt, C.; and Vecchi, M. 1983. Optimization by Simulated Annealing. *Science*, 220, 671-680.

McBride, B. Jena: Implementing the RDF Model and Syntax specification. In *2nd Int'l Semantic Web Workshop*, 2001.

Milgrom, P. 2000. Putting auction theory to work: The simultaneous ascending auction. *Journal of Political Economy.*

L. Miller. Inkling: RDF query using SquishQL. http://swordfish.rdfweb.org/rdfquery.

L. Miller, A. Seaborne, and A. Reggiori. Three implementations of SquishQL, a simple RDF query language. In *First Int'l Semantic Web Conference*, 2002.

National Institute of Standards and Technology. Publication 180-1: Secure hash standard, 1995.

W. Nejdl, B. Wolf, C. Qu, S. Decker, M. S. A. Naeve, M. Nilsson, M. Palmer, and T. Risch. EDUTELLA: A P2P networking infrastructure based on RDF. In *11th World Wide Web Conference*, 2002.

W. Nejdl, M. Wolpers, W. Siberski, C. Schmitz, M. Schlosser, I. Brunkhorst, and A. Lser. Super-peer-based routing and clustering strategies for RDF-based peer-to-peer networks. In *12th World Wide Web Conference*, May 2003.

A. Rao, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica. Load balancing in structured P2P systems. In *2nd Int'l Workshop on P2P Systems*, 2003.

S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content addressable network. In *ACM SIGCOMM*, 2001.

S. Ratnasamy, S. Shenker, and I. Stoica. Routing algorithms for DHTs: Some open questions. In *2nd Int'l Workshop on P2P Systems*, Feb. 2003.

P. Reynolds and A. Vahdat. Efficient peer-to-peer keyword searching. In *ACM/IFIP/USENIX International Middleware Conference(Middleware 2003)*, 2003.

M. Ripeanu, I. Foster, and A. Iamnitchi. Mapping the Gnutella network: Properties of large-scale peer-to-peer systems and implications for system design. *IEEE Internet Computing Journal*, 6(1), 2002.

A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218, 2001.

R.V.Guha. rdfDB : An RDF database. http://guha.com/rdfdb.

Sandholm, T., and Lesser, V. 1997. Issues in Automated Negotiation and Electronic Commerce: Extending the Contract Net Framework. *Readings in Agents*, pp. 66-73, Morgan Kaufmann.

S. Saroiu, P. K. Gummadi, and S. D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Multimedia Computing and Networking*, Jan. 2002.

C. Schmidt and M. Parashar. Flexible information discovery in decentralized distributed systems. In *12th IEEE International Symposium on High Performance Distributed Computing (HPDC'03)*, 2003.

Selman, B.; Kautz, H.; and Cohen, B. 1996. Local search strategies for satisfiability testing. AAAI-92, *DIMACS Series in Discrete Mathematics and Theoretical Computer Science, American Mathematical Society*, 26:521-532.

S. Sen and J. Wong. Analyzing peer-to-peer traffic across large networks. In *ACM SIGCOMM Workshop on Internet Measurement*, Nov. 2002.

G. Singh, S. Bharathi, A. Chervenak, E. Deelman, C. Kesselman, M. Manohar, S. Pail, and L. Pearlman. A metadata catalog service for data intensive applications. In *SC2003 Conference*, November 2003.

Smith, R. 1980. The Contract Net Protocol: High-Level Communication and Control in a

Distributed Problem Solver. *IEEE Transactions on Computers* 29(12):1104-1113.

I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *ACM SIGCOMM*, 2001.

H. Stuckenschmidt, R. Vdovjak, G.-J. Houben, and J. Broekstra. Index structures and algorithms for querying distributed rdf repositories. In *the 14th World Wide Web Conference (WWW2004)*, May 2004.

D. Tam, R. Azimi, and H. A. Jacobsen. Building content-based publish/subscribe systems with distributed hash tables. In *International Workshop On Databases, Information Systems and Peer-to-Peer Computing*, September 2003.

C. Tempich, S. Staab, and A. Wranik. Remindin': Semantic query routing in peer-to-peer networks based on social metaphors. In *the 14th World Wide Web Conference (WWW2004)*, May 2004.

Waldspurger, C., and Weihl, W. 1994. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pp. 1–11.

Walsh, W.; Wellman, M.; Wurman, P.; and MacKie-Mason, J. 1998. Auction protocols for decentralized scheduling. In *Eighteenth International Conference on Distributed Computing Systems*, Amsterdam, The Netherlands.

Walsh, W.; Wellman, M. 1998. Market SAT: An extremely decentralized (but really slow) algorithm for prepositional satisfiability. In *Seventh National Conference in Artificial Intelligence*, 303-309, 2000.

Wellman, M. 1996. The economic approach to artificial intelligence. *ACM Computing Surveys 28* (4es):14–15.B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report CSD-01-1141, UC Berkeley, 2001.

C. T. C, Z. Xu, and S. Dwarkadas. Peer-to-peer information retrieval using self-organizing semantic overlay networks. In *ACM SIGCOMM2003*, 2003.

Accounting Checklist

SEPTEMBER 2004

LCS Input Deadline..Thursday, September 12th
LCS Run...Tuesday, September 17th
Internal JV Deadline..Monday, September 20th
SJ JVs..Thursday, September 23rd
RJ JVs..Friday, September 24th

		FACILITIES BILLING
Mark	☐	Review Facilities spreadsheet to ensure correct allocation of FAC income/charges
Mark	☐	Input all Facilities JVs
		IPC BILLING
Mark	☐	Update entities table for NH #'s
Facilities	☐	Update entities/visitors table for any new visitors
Human Resources	☐	Update entities table for new hires
Action	☐	Provide new and deleted login Ids during month as received. Send notification to Mark.
Action	☐	Provide new home access Ids during month as received and also as deleted. Send Mark notification of any changes.
Mark	☐	Prepare IPC retro JVs
Mark	☐	Distribute IPC alpha charges report to Contracts/Accounting
Mark	☐	Prepare IPC JV report and send final reports via e-mail to the division directors.
Mark	☐	Input all IPC JVs
		VACATION BILLING
Mark	☐	Prepare and input vacation accrual JVs

Mark	☐	Prepare vacation used calculation
Mark	☐	Prepare vacation used, retro and payout JVs
		COMMON BILLING
Eloisa/Mark	☐	Update common worksheet spreadsheet: 1. Add new accounts, 2. Delete expired accounts, 3. Input salary data.
Eloisa/Mark	☐	Prepare and input monthly common billing JV
		MISC BILLINGS
Monica	☐	Prepare and input Ralph's market, Remedy, FedEX JVs
Mark	☐	Prepare and input MOSIS Fab runs, Los Nettos, VR Facility, and other cost transfer JVs
Raquel Rios	☐	Create monthly JV binder
Casandra	☐	Review and verify all JVs for accuracy
		DIVISION ANALYSIS (FORECASTING)
Casandra, Lisa and Raquel	☐	Prepare and review monthly division analyses and ad-hoc reports. (Casandra 2 & 3,--Raquel 4 & 7, --Lisa 1, 8, 9 & 10).
		REPORTING
Casandra/Raquel	☐	Prepare IPC Income and Expense Reports ☐ East ☐ West (Send e-mail for IPC West (see special format) and IPC East)
Casandra/Raquel	☐	Prepare ISI Facilities Income and Expense Reports ☐ East ☐ West (Send e-mail for FAC East)

		REPORTING (cont.)
Casandra	☐	Reconcile Equipment Depreciation Account (22-1540-2626/22-1540-4835)
Casandra/Raquel	☐	Prepare Common Income and Expenses Report ☐ East ☐ West (Send e-mail for Common East)
Casandra/Raquel	☐	Prepare and send e-mails for DIV ADMIN Income and Expenses Report ☐ Div 1 ☐ Div 2 ☐ Div 3 ☐ Div 4 ☐ Div 7 ☐ Div 8
Casandra/Raquel	☐	Prepare Virtual Reality (send e-mail) ISI East Hardware Clearing Analysis
Casandra	☐	Submit FIMS data: ACT Templates: ***The FIMS reports should be completed and e-mailed by the 5th of each month to C.Porter, Project Leader, Div. Admin. and Beverly.*** ACT-TEMP 53-4540-0505/53-4540-0506/53-4533-0506/53-4540-0510, TEMPLE 53-4540-4224 TASK: TASK 99 (53-4540-1157 , 53-4509-1158 &53-4540-1159) and WEBSCRIPTER (53-4540-7734 and 53-4540-7735), DASADA: SIMS-TABASSCO (53-4540-0611, 53-4540-0612); EELD: KOJAK (53-4540-0584/53-4540-0585/53-4540-0586/53-4509-0586/53-4540-0587 and 53-4540-0588) (***Note at bottom of e-mail break out Rap Teams and KOJAK/MOJAK**); WIDELINK (53-4540-0193/53-4540-0194/53-4540-0195/53-4540-0196)
Casandra	☐	Prepare Monthly JIST Analysis (53-4540-2301-Send to Beverly)
Casandra	☐	Prepare NASA IPG NPACI **Quarterly** Report (53-4540-4241, 53-4540-4240 and 53-4540-4242) Prepare NPACI Quarterly Report (53-4540-1017/53-4540-1018)
Casandra	☐ ☐	Prepare Unrestricted Report (monthly) Prepare UNR Allocation Report (quarterly)
Casandra	☐	Prepare monthly financial package for financial meeting (powerpoint slides, obligated report, account summary, cost sharing, labor spread and award summary)
Raquel	☐	Prepare MOSIS Clearing analysis (for Cesar)
Raquel	☐	Prepare MOSIS EMS02 and EMS02 T&M Task Analysis (for Gwen)
Lisa	☐	Prepare Kesselman Analysis
		PROCUREMENT CARDS
Raquel	☐	On-line updating of clearing account charges

		PROCUREMENT CARDS (cont.)
Raquel	☐	Prepare manual JVs of clearing account charges
Raquel Rios	☐	Distribute statements to cardholders
Raquel Rios	☐	Reconcile statements with receipts and supporting documentation
Raquel Rios	☐	Maintain spreadsheets of cardholders and outstanding statements
		MISCELLANEOUS
Raquel	☐	ISI East Petty Cash Reimbursements
Raquel	☐	Invoice Third Parties--FAST Exchange, Fetch, ICANN for suite rental and IPC costs
Casandra	☐	Reconcile NOREC account (12-1540-0010)
Casandra	☐	Review program recharge centers and master accounts for correctness
Raquel	☐	Reconcile procard clearing account (12-1540-5405)
Raquel	☐	Reconcile and Review travel clearing account (12-1540-0018)
Raquel Rios/Raquel	☐	Deposit reimbursement (shipping, telephone, postage, paper, copying, etc.). MOSIS and Los Nettos checks
Lisa	☐	Review Tuition Billing Report